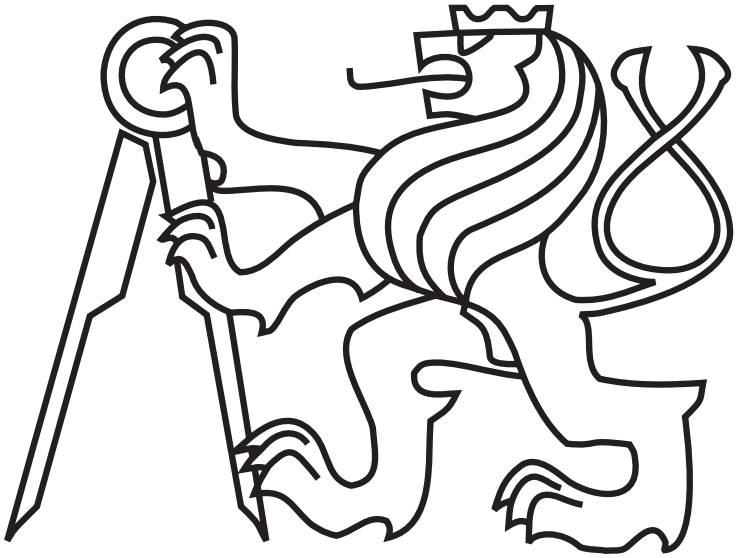
CZECH TECHNICAL UNIVERSITY IN PRAGUE

Faculty of Electrical Engineering

BACHELOR THESIS



Vojtěch Lhotský

Demonstration Tasks for the SyRoTek System

Department of Cybernetics

Thesis supervisor: **RNDr. Miroslav Kulich, Ph.D.**

České vysoké učení technické v Praze
Fakulta elektrotechnická

Katedra kybernetiky

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Student: Vojtěch L h o t s k ý

Studijní program: Kybernetika a robotika (bakalářský)

Obor: Robotika

Název tématu: Demonstrační úlohy pro systém SyRoTek

Pokyny pro vypracování:

SyRoTek je systém pro vzdálenou výuku mobilní robotiky a příbuzných oborů vyvinutý na Katedře kybernetiky a umístěný v laboratoři E132. Cílem práce je seznámit se s tímto systémem a implementovat sadu demonstračních úloh, které budou demonstrovat základní funkcionality systému a které budoucím uživatelům systému zpříjemní první kroky s ním. Při vypracování postupujte podle následujících kroků:

1. Seznamte se se systémy SyRoTek (http://syrotek.felk.cvut.cz) a ROS (Robot Operating system, http://ros.org) a jejich propojením.
2. Naimplementujte vybrané demostrační úlohy (seznam konzultujte s vedoucím práce).
3. Funkčnost úloh demonstrujte nejprve v simulátoru a poté i na reálných robotech systému SyRoTek.
4. Z demonstrací natočte video a úlohy důkladně zdokumentujte.

Seznam odborné literatury:

[1] Kulich, M.; Chudoba, J.; Kosnar, K.; Krajnik, T.; Faigl, J.; Preucil, L.: "SyRoTek—Distance Teaching of Mobile Robotics," Education, IEEE Transactions on , vol.56, no.1, pp.18,23, Feb. 2013.
[2] Siciliano, B.; Khatib, O. (Eds.): Springer Handbook of Robotics. Springer 2008. ISBN 978-3-540-23957-4.
[3] H. Choset, K. M. Lynch, S. Hutchinson, G. Kantor, W. Burgard, L. E. Kavraki and S. Thrun: Principles of Robot Motion: Theory, Algorithms, and Implementations, MIT Press, Boston, 2005.

Vedoucí bakalářské práce: RNDr. Miroslav Kulich, Ph.D.

Platnost zadání: do konce letního semestru 2014/2015

L.S.

doc. Dr. Ing. Jan Kybic
vedoucí katedry

prof. Ing. Pavel Ripka, CSc.
děkan

V Praze dne 10. 1. 2014

Czech Technical University in Prague
Faculty of Electrical Engineering

Department of Cybernetics

BACHELOR PROJECT ASSIGNMENT

Student: Vojtěch L h o t s k ý

Study programme: Cybernetics and Robotics

Specialisation: Robotics

Title of Bachelor Project: Demonstration Tasks for the SyRoTek System

Guidelines:

SyRoTek is a system for distant learning of mobile robotics and related areas developed at Department of Cybernetics and placed at E-132 laboratory. The aim of thesis is to get acquainted with this system and to implement a set of demonstration tasks that will demonstrate fundamental functionalities of the systems and that will make the first steps with the system easier for the future users. Keep the following guidelines:

1. Get acquainted with the SyRoTek system (http://syrotek.felk.cvut.cz) and ROS framework (Robot Operating system, http://ros.org) as well as with their interconnection.
2. Implement selected demonstration tasks (consult the list with the supervisor).
3. Demonstrate the functionality of the tasks in the simulator and then also with real robots on the SyRoTek system.
4. Record videos of the demonstrations and document the tasks carefully.

Bibliography/Sources:

[1] Kulich, M.; Chudoba, J.; Kosnar, K.; Krajnik, T.; Faigl, J.; Preucil, L.: "SyRoTek—Distance Teaching of Mobile Robotics," Education, IEEE Transactions on , vol.56, no.1, pp.18,23, Feb. 2013.
[2] Siciliano, B.; Khatib, O. (Eds.): Springer Handbook of Robotics. Springer 2008. ISBN 978-3-540-23957-4.
[3] H. Choset, K. M. Lynch, S. Hutchinson, G. Kantor, W. Burgard, L. E. Kavraki and S. Thrun: Principles of Robot Motion: Theory, Algorithms, and Implementations, MIT Press, Boston, 2005.

Bachelor Project Supervisor: RNDr. Miroslav Kulich, Ph.D.

Valid until: the end of the summer semester of academic year 2014/2015

L.S.

doc. Dr. Ing. Jan Kybic
Head of Department

prof. Ing. Pavel Ripka, CSc.
Dean

Prague, January 10, 2014

Prohlášení autora práce

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze dne..... .....

Podpis autora práce

Acknowledgements

I would like to thank my supervisor RNDr. Miroslav Kulich, Ph.D. for his advice and guidance throughout this project.

Abstrakt

Cílem této práce je vytvořit praktický návod pro uživatele systému SyRoTek. Tento návod je vytvořen formou sady demonstračních úloh implementovaných v ROSu (Robotický Operační Systém). Tato práce obsahuje popis práce se SyRoTkem, ROSem a všech vytvořených demonstračních úloh. Každá úloha byla otestována na skutečném robotu a všechny výsledky jsou popsány v této práci.

Abstract

The goal of this thesis is to create a practical guide for the users of SyRoTek system. The guide is in the form of a set of demonstration tasks implemented in the ROS (Robot Operating System). This thesis contains a tutorial for the work with the SyRoTek, the ROS, and a description of all created demonstration tasks. Every task was tested on a real robot and all the results are published in this thesis.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

SyRoTek is a system for distant learning of mobile robotics and related areas developed by the Department of Cybernetics and located at E-132 laboratory. The system consists from an Arena with dimensions of 3.5×3.8 m and mobile robots. Each robot has several sensors (including laser rangefinder, sonar or accelerometer) and can be charged in docking stations within the Arena. The user can access the SyRoTek via internet and use it for developing and testing robotic (or multi-robotic) applications. Each robot has a large set of sensors onboard and there are cameras on top of the arena as well. That allows the user to create a wide variety of tasks.

The SyRoTek has been used in courses taught at the CTU in Prague (*Introduction to Mobile Robotics* and the *Practical Robotics*) and the University of Buenos Aires. In the time of writing [3] about ten students already used SyRoTek during their thesis in the areas of multi-robot exploration, formation control and swarm robotics [3]. For example, the SyRoTek was recently used to experiment with autonomous snow shoveling of an airport model.

There are manuals explaining the work with the SyRoTek but they are obsolete and not complete as the system is being continously developed, moreover they do not contain a guide that would show future users, how to write applications and test them on practical examples. The aim of this work is to create a set of demonstration tasks in which the basic functions of the SyRoTek are shown. This thesis can serve as a guide for beginner users and help them with creating their own robotic applications.

All tasks are implemented in the ROS[1] framework. The C++ programming language is used because it has probably the best support by ROS. The tasks show main functionalities of the SyRoTek and the ROS as well as their interconnection.

At the beginning of the thesis the basics of working with the ROS (Section 2.1) and the SyRoTek (Section 2.2), simulating robot's behaviour in a Stage simulator[2] (Section 2.3) and visualising data in the Rviz[3] (Section 2.4) are explained .

[1] ROS - Robot Operating System, http://wiki.ros.org/

[2] Stage is 2D multiple - robot simulator, http://playerstage.sourceforge.net/?src=stage

[3] Rviz is 3D visualization tool for the ROS, http://http://wiki.ros.org/rviz

This is followed by a description of particular demonstration tasks in the Chapter 3. The initial tasks are very simple applications. They teach the user the basics of the communication with the robot. The difficulty of the tasks increases gradually, so that even more experienced users can find some inspiration for their own applications.

The ROS system has many interesting features for more efficient work. This thesis mentions parameter server and launch files (Chapter 4). More attention is also devoted to transformations and problems with their timing (Chapter 5). The transformations and the parameter server are used in an autonomous exploration. The goal of this task is to create a map of an unknown environment (using external localization).

Finally 2D navigation stack[4] and Gmapping[5] packages are used with a part of the original exploration task to run an autonomous exploration with its own localization. It uses the data from the laser rangefinder and the odometry measured by the robot itself (Chapter 7).

This paper contains only shortened explanation of demonstration tasks. Complete explanation is attached on the CD.

[4]The 2D Navigation Stack safely guides the robot to the target position, http://wiki.ros.org/navigation
[5]The Gmapping creates a map and provides a localization of the robot, http://wiki.ros.org/gmapping

Chapter 2

ROS, SyRoTek and simulator

This chapter is not a complete guide to ROS or SyRoTek as good sources exist [4][5]. Instead, it aims to provide first aid to users with some experience with these systems. The main goal of this chapter is to explain the necessary parts of ROS and SyRoTek for the following demonstration tasks.

2.1 ROS (Robot Operating System)

The Robot Operating System (ROS) is a flexible framework for writing robot software. It is a collection of tools, libraries, and conventions that aim to simplify the task of creating complex and robust robot behavior across a wide variety of robotic platforms [6].

ROS is officialy supported only on Ubuntu, but runs on many operating systems (other Linux distributions, Windows, OS X). Groovy Galapagos on Ubuntu 12.04 LTS has been used in this work.

Each process running under the ROS can be interpreted as a graph node. These processes are loosely coupled by ROS communication infrastructure [7]. Nodes in ROS use ROS client libraries, which allow communication of nodes written in different programming languages. Nodes can communicate by using topics (send messages to a topic or subscribe a topic to receive messages, more information in Section 2.1.5). A node can also provide or use services [8]. ROS has one master node, which provides name service - helps the other nodes to find each other. It starts, when the roscore is launched. It can be done by using command:

```
roscore
```

Other nodes could be started by launch file[1] or by using command:

```
rosrun <package_name> <executable_name> <parameters>
```

[1]ROS launch file is a file written in xml that can launch one or multiple nodes (more in Section 4.2).

2.1.1 Creating packages

To create programs for ROS, it is necessary to prepare a package first. In ROS, there are two ways how to create and compile packages. The first one is through `rosbuild`, which was used in older versions of ROS and is still supported. Another tool – `catkin` – is used from the Groovy Galapagos version. It provides easier work on different operating systems, easier compilation of large projects, faster build and other improvements [9]. The `catkin` will be used in this thesis (therefore the programs will not be able to compile under older versions of ROS than Groovy Galapagos).

To create a package in `catkin`, open your cakin workspace first[2]. Then move to `src/` directory. Following command will create catkin package [10]:

```
catkin_create_pkg <package_name> [dep1] ... [depN]
```

This command creates a new directory named `<package_name>` in `src/` and sets dependence on another packages. Almost every application in this thesis depends on packages `roscpp` and `std_msgs`. The `roscpp` allows you to write applications in C++ language (for writing in python you can use `rospy`, other languages are not recommended because they do not have so extensive support). The package `std_msgs` lets you to use standart ROS messages. There are other packages with messages (for example `geometry_msgs`), but it is best to use them only if you need them.

As a dependence on the package `roscpp` is included, there are 2 files in the new directory (`package.xml`, `CMakeLists.txt`) and 2 directories (`src/`, `include/`). In `package.xml` you can fill information about package (author, version, name, description, ...). Place header files into the `/include` directory and source files into `src/`.

2.1.2 Preparing `CMakeLists.txt`

In order to compile program, you need to set up `CMakeLists.txt`. Usually, it will be only few lines at the end of the file. First you must tell, which files need to be compiled. You can create variable (in this case `SRCS1`), which contains all necessary source files. Add following lines to the end of `CMakeLists.txt`:

```
set (SRCS1 ${SRCS1} src/file_1.cpp)
set (SRCS1 ${SRCS1} src/file_2.cpp)
```

Next include catkin directories by the following line:

```
include_directories(include ${catkin_INCLUDE_DIRS})
```

Now create executable and link catkin libraries to it.

[2] If you do not have prepared catkin workspace, follow this tutorial: http://wiki.ros.org/ROS/Tutorials/InstallingandConfiguringROSEnvironment

```
add_executable(executable_name ${SRCS1})
target_link_libraries(executable_name ${catkin_LIBRARIES})
```

Everything is now prepared and you can start programming [10].

2.1.3 Building a program

When the programming is done, open your catkin workspace in a terminal and call:

```
catkin_make
```

This command compiles all packages in your catkin workspace. If there are problems with compilation, make sure, that you have a path to your catkin workspace in `ROS_PACKAGE_PATH` variable. If you do not, export it by using command:

```
export ROS_PACKAGE_PATH+=":<path_to_your_catkin_workspace>/src/"
```

If the `catkin_make` fails after processing packages and returns error: `make: illegal option -- l`, you should be able to compile the project by calling:

```
catkin_make --cmake -j24
```

The `catkin_make` uses by default parameters `-j24`[3] and `-l24`[4]. If the parameter `-l24` is not accepted by compiler, use only `-j24`.

2.1.4 ROS and C++

The C++ language was chosen for this thesis and all programs will be written in it. To write programs in C++, the package must have `roscpp` in its dependencies. First include all the headers necessary for common pieces of the ROS system [11]:

```
#include "ros/ros.h"
```

Before any part of the ROS system can be used, you must do initialization:

```
ros::init(argc, argv, "name_of_the_node");
```

[3]The `-j` parameter tells make to run multiple parallel processes and the integer defines how many processes can be executed at once.

[4]The `-l` parameter limits the number of jobs to run at once based on the load average. The floating-point number after the parameter sets maximal load (if the load is higher, make will not start parallel processes).

The `ros::init()` function needs `argc` and `argv` to process ROS arguments and remap names according to parameters set in the command line. Usage of ROS parameters will be explained in Section 4.1. When ROS is initialized, create NodeHandle, which manages communications between the node and the ROS system [11].

```
ros::NodeHandle n;
```

When all this is done it is possible to work with the ROS system.

2.1.5 Messages and topics

As mentioned before, communication between separate nodes is managed through topics. Every node can publish messages (of the right type) to that topic and every node can receive those messages by subscribing. There are many default type of messages in ROS and it is possible to create new types as well. As was mentioned in Chapter 2.1.1, basic messages are in the `std_msgs` package. Most of messages used in this thesis will contain `std_msgs::Header` so the princip will be explained on that example. You can find out more information about this type of message by using command:

```
rosmsg show Header
```

The result is:

```
[std_msgs/Header]:
  uint32 seq
  time stamp
  string frame_id
```

The message contains three variables. Variable `seq` represents the sequence number (if you are sending multiple messages, this variable helps to keep them in the right order). The `stamp` represents time in epoch[5] format (usually when the message was created) and the `frame_id` will be explained with transformations (Chapter 5). This message can be accessed as the `std_msgs::Header` class in C++.

To publish messages, you must create a publisher object first:

```
ros::Publisher pub = n.advertise<std_msgs::Header>("topic_name", 1000);
```

This creates the publisher object for Header messages. The number 1000 represents a size of the buffer[6]. Now you can prepare messages and send them by `publish` function:

```
pub.publish(message);
```

[5] http://www.epochconverter.com/

[6] Each publisher and subscriber has a buffer that contains predefined number of previous messages.

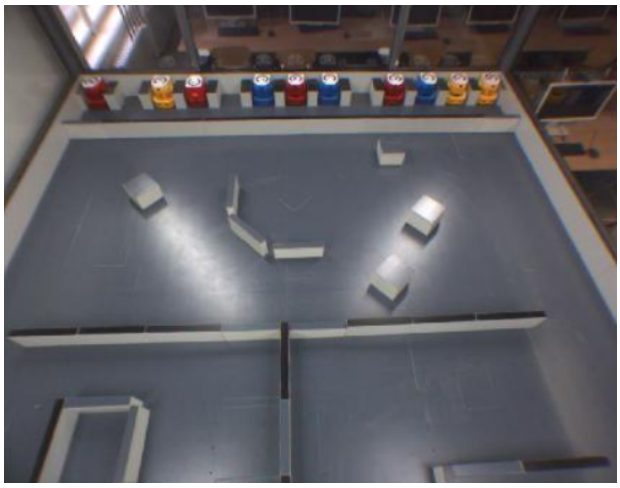


Figure 2.1: The SyRoTek arena (view from the camera above)

When subscribing to a topic, create callback function. It should have as a parameter constant pointer to the right type of message, for example:

```
void callback(const std_msgs::Header::ConstPtr& msg);
```

Now create a subscriber. It calls this function each time when some message is published to the topic.

```
ros::Subscriber sub = n.subscribe("topic_name", 1000, callback);
```

The first two parameteres are the same as for the publisher and the last one is a name of the function to be called. It is possible to call a method of a class instance:

```
ros::Subscriber sub = n.subscribe("topic_name", 1000, &SomeClass::callback
   , instanceOfThatClass);
```

Finally use the following line to start subscribing messages:

```
ros::spin();
```

2.2 SyRoTek

The SyRoTek ("System for RoboTic E-learning") allows you to remotely (via internet) control a multi-robot platform in a dynamic environment [5]. The SyRoTek has an Arena with dimensions of 3.5×3.8 m. In the Arena there are 13 docking stations, where robots can charge. There are reconfigurable obstacles (some could be controlled remotely, others are fixed). Several cameras are placed above the Arena. SyRoTek has localization system based on processing images from the camera above the arena. This localization provides global odometry. The robots

in SyRoTek arena are called S1R. They have dimensions $174 \times 163 \times 180$ mm. Each robot can also measure locally its own position (local odometry). The S1R robot has many sensors, but the most important for this thesis is a laser rangefinder which measures distance to obstacles around robot [12].

The SyRoTek is world-wide available platform, but due to limited capacity, the access to robots is for registred users only. If you are not registred yet, you are requested to fill the registration form at the SyRoTek main page[7]. There (after you login) click on the Courses section. Enroll some course first (*Demos and tasks of IMR members* course is used in this thesis), open it and choose a task (*One robot with a laser rangefinder* is compatible with all demonstration tasks). Finally you can create a reservation, which allows you to control the robot [5].

2.2.1 Running ROS on SyRoTek

Let's assume you have an active reservation and one robot with laser rangefinder (the guide to work with SyRoTek is in the "Practical Guide to the Syrotek System" [5]). Connect to the SyRoTek via `ssh` and use the following command to launch `roscore` and nodes to operate the robot:

```
roslaunch /opt/syros/syros.launch
```

If there are problems with `GLIBCXX`, one of the solution is:

```
export LD_LIBRARY_PATH=/usr/local/lib/gcc46:$LD_LIBRARY_PATH
```

When the roscore is running, let's look at the topics provided by SyRoTek.

2.2.2 Topics and messages from SyRoTek

To look at all the topics use command:

```
rostopic list
```

The result is:

```
/rosout
/rosout_agg
/syros/base_cmd_vel
/syros/base_odom
/syros/base_pose
/syros/global_cmd_vel
```

[7] https://syrotek.felk.cvut.cz/

```
/syros/global_odom
/syros/global_pose
/syros/laser_laser
/tf
```

From these topics only the following ones will be necessary for demonstration tasks:

- /syros/base_cmd_vel - This is the topic for controlling robot's movement.
- /syros/base_odom - This topic provides messages with local odometry (measured by robot)
- /syros/global_odom - Global odometry from the localization system
- /syros/laser_laser - Data from the laser scan
- /tf - Transformations

To obtain detailed information about a topic (including type of message) use command:

```
rostopic info <topic_name>
```

2.3 Stage simulator

The behaviour of a robot in the SyRoTek arena could be simulated in Stage simulator. In the file `eci-arena.world` is a world file with the model of SyRoTek Arena. The original file was modified to work with the version for ROS. The simulator could be run by using the following command:

```
rosrun stage stageros <path_to_the_world_file>/eci-arena.world
```

After executing this command, a window should appear (as in Fig. 2.2). If you would like to change some things in the Arena, you can simply drag obstacles or the robot using a mouse.

2.3.1 Topics and messages from the simulator

```
/base_pose_ground_truth
/base_scan
/clock
/cmd_vel
/odom
/rosout
/rosout_agg
/tf
```

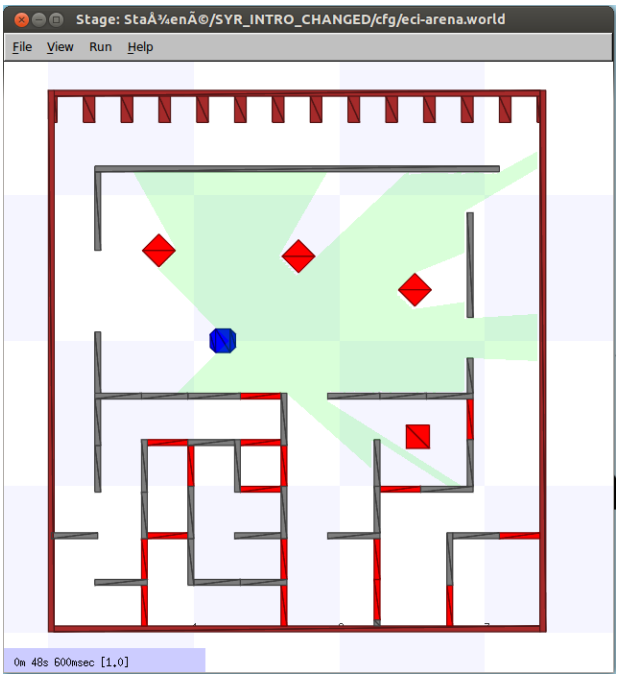


Figure 2.2: Stage Simulator

If you look at the topics, you can see that there is no global or local odometry just `/odom`. The simulator has only one odometry, which is usually ideal (it is possible to set up some odometry error in the world file). The data from a laser rangefinder are published to the `/base_scan` topic and the `/cmd_vel` topic is used for controlling robot's movement.

2.4 Rviz

Rviz is a visualization system in the ROS. You can view odometry, transformations, map, path and many more. Use the following command to start rviz [13]:

```
rosrun rviz rviz
```

After running this command a windows should appear similar to Fig. 2.3. Set fixed frame to `/odom` in the simulator or `/arena` in the SyRoTek. Press add to set up what should be viewed. By default, you should add a grid with the same reference frame as the fixed frame in global options. Add TF next. That should show the transformation tree (see more about transformations in Chapter 5).

To visualize data from the laser rangefinder add it as well and set its frame to `/base_scan` in the simulator or `/syros/laser_laser` in the SyRoTek.

If there are problems with visualizing data from the SyRoTek in RVIZ, it might be caused by bad time sychronization. Some versions of the RVIZ also crash when try to view data from

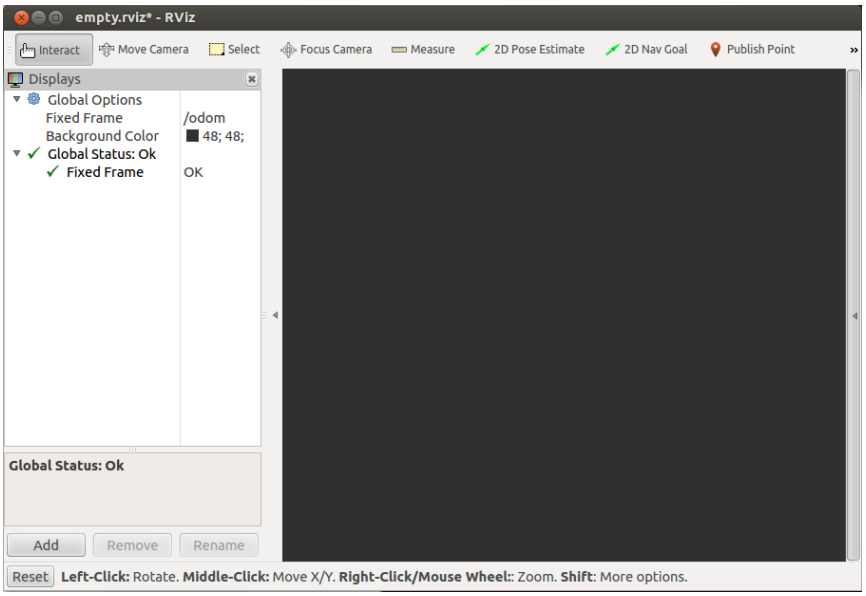


Figure 2.3: Rviz empty

the laser rangefinder.

Chapter 3

The demonstration tasks

This chapter focuses on the simpler demonstration tasks. On these tasks are shown the basics of working with ROS and SyRoTek. Processing the data from the laser rangefinder and publishing `Twist` messages (commands for robot) are explained in the Braitenberg vehicle task (Section 3.1). The work with odometries is shown in PID controller and Dead Reckoning (Sections 3.2 and 3.3). The Wall Following and the Trajectory Following tasks are slightely more complex, but they still use only the basics of the ROS system.

3.1 Braitenberg vehicle

The goal of this task is to show how to write a simple application in ROS. Braitenberg vehicle uses a very simple algorithm so it is good to start with it.

The Braitenberg vehicle is an autonomous vehicle. It usually has two primitive sensors, which are directly connected to the motors. It means, that if the right sensor sends higher values, the right motor runs faster and the robot turns left. By this simple algorithm an obstacle avoidance behavior can be achieved [14].

Assume, you have a robot with only a laser scanner, which measures distances from the robot to obstacles. To simulate behavior of the Braitenberg vehicle, the algorithm finds the minimal distances on the left and right side (the closest obstacles). You can not simply set up the left and right motor to follow these values as ROS allows to set up linear velocity and angular velocity, so some conversion is needed (see bellow).

3.1.1 Processing inputs and outputs

Vehicle behavior is implemented in the class `NodeBraitenberg2`. This class has two methods (except constructor and destructor). The first method processes laser scan data and the second one controls the robot.

Data from laser rangefinder are in message type `LaserScan`. It contains the following variables:

```
std_msgs/Header header
float32 angle_min
float32 angle_max
float32 angle_increment
float32 time_increment
float32 scan_time
float32 range_min
float32 range_max
float32[] ranges
float32[] intensities
```

Vector `float32[] ranges` contains raw data from the laser scanner. As mentioned above, you can simulate Braitenberg vehicle by finding the minimal distances on the left and right side. Assuming that the sensor is positioned symmetrically, you can find minimum value from the first and second half of the `ranges` vector and use those values as the minimum on the left and right side. Knowing angles corresponding to the minimum values might be useful. If you know the index of the minimal value in the `ranges`, you can calculate angle by using this equation:

$$\varphi = \left(i - \frac{l}{2}\right) \cdot \theta, \tag{3.1}$$

where φ is angle of i-th element of vector, l is lenght of the vector and θ is angle incrementation (`angle_increment`).

Commands for robot are sent through the Twist messages:

```
geometry_msgs/Vector3 linear
geometry_msgs/Vector3 angular
```

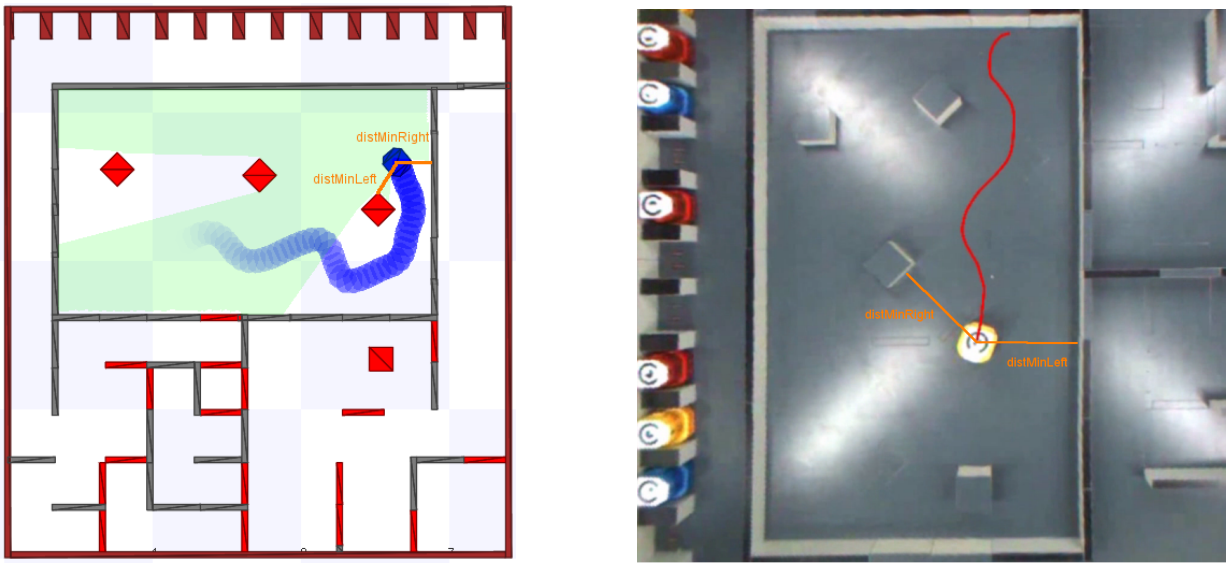
The Twist message contains two 3D vectors[1] for linear and angular velocity, but the robot moves only in 2 dimensional space so `x` in linear (for linear velocity) and `z` in angular (for angular velocity) will suffice. You can leave other values equal to zero.

3.1.2 Conversion

Assuming that you have the minimum values, the only problematic part is to make conversion of these values to linear and angular velocities. Because the linear velocity has no effect (theoretically) on the direction of movement, it can be set to a constant value. You can use the following equation to calculate the angular velocity:

$$|\omega| = c \cdot \frac{d_{long}}{d_{short}} - c,$$

[1] The `geometry_msgs/Vector3` has elements `float64 x,y,z`.



(a) The experiment in simulator

(b) The experiment in SyRoTek arena

Figure 3.1: The Braitenberg vehicle

where

$\|\omega\|$	is the absolute value of the angular velocity,
c	is a constant coeficient representing sensitivity of the ratio of the minimal distances,
d_{long}	is the greater value from the minimal distances on the left and right side and
d_{short}	is the smaller value from the minimal distances on the left and right side.

As you can see, this formula provides the absolute value of the angular velocity only. That will be sufficient, because the sign is determined by a simple `if/else` construction.

You can now process input data, make conversion to linear and angular velocity and send commands to the robot.

3.1.3 Experiments

The Braitenberg vehicle was first tried in the simulator. As you can see in Fig. 3.1a, the robot is succesfully avoiding obstacles. It keeps the minimal distance on the left equal to the minimal distance on the right.

The Braitenberg vehicle was also tested in the real Arena (Fig. 3.1b). The robot in SyRoTek balances distances slower, which causes going closer to the obstacles and sharper turns afterwards. The video record of this experiment can be found on the SyRoTek website[2].

[2]https://syrotek.felk.cvut.cz/about/videos

3.2 PID Controller in ROS

The aim of this tutorial is to show how to write a PID[3] controller in ROS. It controlls the position and orientation of a robot. The robot should be able to move forward x meters where x is a reference value or to turn around by φ radians where φ is a reference value.

Input to the PID controller are global odometry messages. First you need to calculate the difference between the reference value and the actual value. Next thing to do is calculating intervention from the PSD[4] controller. All of this is implemented in the `NodePID` class. Input is gathered by using message callback.

3.2.1 Processing inputs and outputs

Commands for the robot are sent through the `Twist` messages. You can find more information about this type of message in Section 3.1.1.

Input to this program is global odometry, which uses messages called Odometry:

```
std_msgs/Header header
string child_frame_id
geometry_msgs/PoseWithCovariance pose
  geometry_msgs/Pose pose
    geometry_msgs/Point position
    geometry_msgs/Quaternion orientation
  float64[36] covariance
geometry_msgs/TwistWithCovariance twist
  geometry_msgs/Twist twist
  float64[36] covariance
```

You need only information about actual position and time. Position in x and y axis is stored in `pose.pose.position`[5]. Time of measurement is in `header.stamp`. The only problem is to get the orientation of the robot. As you can see, there is a quaternion called `orientation`[6]. This quaternion is used to describe 3D rotation, but the robot operates only in 2D space. The elements `x`, `y` and `z` in a quaternion are related to rotation about these axes. Rotation in 2D space is the same as rotation about z axis. Relation between the angle (φ) and z element in the (Q_z) quaternion in 2D is as follows [15]:

$$\varphi = 2 \cdot \arcsin(Q_z)$$

[3]The PID controller contains three parts (proportional, integral and derivative). Sum of these parts creates an action intervention (see equation 3.3).

[4]PSD is a discrete version of the PID controller. The integral part is replaced by a sum (see equation 3.4).

[5]The `geometry_msgs/Point` structure contains elements `float64 x,y,z`.

[6]The `geometry_msgs/Quaternion` contains elements `float64 x,y,z,w`.

3.2.2 Angle and distance

To calculate the difference between reference and actual values, you need to calculate distance and angle between the start and actual position (distance and angle between two points).

$$d = \sqrt{(y_2 - y_1)^2 + (x_2 - x_1)^2} \tag{3.2}$$

$$\theta = atan2(y_2 - y_1, x_2 - x_1)$$

To simplify the code in `NodePID` class create class `MyPoint`, which stores data about position and has methods to calculate the distance and the angle.

3.2.3 Errors

To calculate the error (difference between the reference and the actual value) you need to calculate the actual value. For rotation it is simple:

$$\varphi_a = \varphi_o - \varphi_s,$$

where φ_a is the actual value, φ_o is the value measured from odometry and φ_s is the angle measured at the start of the program. For translation you need to calculate a distance between the start and the actual position, which can be done by using equation 3.2.

3.2.4 PID (PSD) controller

You can imagine PID controller as a sum of P-controller, I-controller and D-controller:

$$v_A(t) = k_P e(t) + k_D \frac{de(t)}{dt} + k_I \int_0^t e(t)dt, \tag{3.3}$$

where

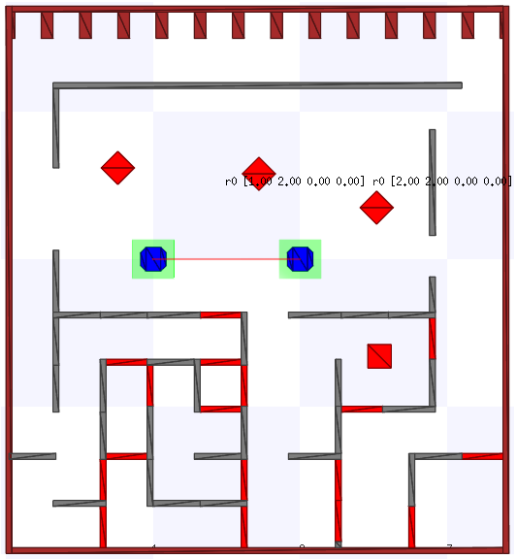
$x_A(t)$ is intervention from PID controller,

$e(t)$ is error,

k_P is proporcional constant,

k_D is derivative constant and

k_I is integral constant.



(a) The experiment in the simulator (forward by 1.0 meters)

(b) The PID controller in SyRoTek

Figure 3.2: The PID controller

Which could be transfered to discrete spectrum (PSD):

$$v_A(t_n) = k_P e(t_n) + k_D \frac{e(t_n) - e(t_{n-1})}{t_n - t_{n-1}} + k_S \sum_{i=0}^{n} e(t_i) \cdot (t_n - t_{n-1}), \tag{3.4}$$

where t_n is the time of n-th iteration and k_S is sum constant (corresponds to k_I in PID).

The same equation can be used for controlling translation as well as rotation. Each controller will have it's own constants. Results from the translation controller and the rotation controller are sent through the `Twist` message (see more in Section 3.1.1) as commands for the robot (translation and rotation velocity).

Constants used in equation 3.4 were obtained by creating a model of the robot (transfer function) and by using an appropriate function in Matlab. If you use SyRoTek or Stage simulator with a model of SyRoTek Arena, you can use constants predefined in the code. Complete explanation is in appendix of the file `syrotek_tutorials.pdf` on the CD.

3.2.5 Experiments

The PID controller uses two arguments. The first is `-f` or `-r` for movement forward or rotation. The second is value which is in meters for movement forward or radians for rotation. As you can see in Fig. 3.2a, the robot was set to move forward by $1.0m$. The robot moved from position $[1.00, 2.00]$ to position $[2.00, 2.00]$. Tolerance was set to $0.01m$ and it was maintained. In the second experiment the robot was set to rotate by $\pi/2$ radians with tolerance 0.02 radians.

It turned by 1.556 radians, which is smaller than $\pi/2$ by 0.015 radians so the tolerance was also maintained.

When testing the application in SyRoTek, robot was set to move forward by 1 meter, turn around by π radians and this was repeated 8 times. In the Fig. 3.2b you can see the deviation from the start position. The tolerance in translation movement was set to 0.01 meters and the tolerance in rotation movement to 0.04 radians.

Behaviour of this algorithm depens on precision and frequency of odometry measurement and on constants for the controller. Tolerance should not be set to be more precise than odometry, because the robot might never achieve the reference value. The video record of this experiment can be found on the SyRoTek website[7].

3.3 Dead Reckoning

The aim of this tutorial is to write a program that measures the error of the odometry. The robot goes several times through a simple geometric shape (in this case square) and at the end, it compares the start position with the end position (which should be ideally the same) and the position in local odometry (measured by the robot) with the position in global odometry (provided by the localization system).

Note: Stageros simulator provides only one odometry (which is ideally precise) so it is not be possible to test this program in a simulator.

The Dead Reckoning is implemented in the `NodeDeadReckoningI` class. Before you start sending commands to the robot, you need to know the start position from local and global odometry. Until both of these positions are received, the robot should not move. To ensure that, create two boolean variables (`globalOdomReceived` and `localOdomReceived`). After both messages are received, start movement.

To follow square trajectory, you need commands like `''move x meters''` and `''turn around by x radians''`, but commands for the robot are sent through `Twist` messages, which allow only to set up angular and linear velocities. You can use PID (PSD) controller that was explained in the previous tutorial. The only change in the controller will be in usage of local odometry instead of global one.

After the robot stops its movement, it waits again for local and global odometry and calculates differences from the start position.

3.3.1 Following square trajectory

When odometry for the start position was received, call method `commander()`, which starts repeatedly calling the PID controller (move forward by x meters and than turn around by $\pi/2$

[7] https://syrotek.felk.cvut.cz/about/videos

radians). The robot should make several squares this way. After this is done, set boolean variable evaluate to `true` and wait for the odometry data.

3.3.2 Evaluating results

After both odometry data are received, call the `evaluateResults()` method. To calculate the difference between the end and start positions you need only to calculate the difference in x, y and φ. A problem occurs when you need to calculate the difference between local and global odometry. The coordinate system of local odometry is shifted and rotated relative to the global odometry.

In both coordinate systems you need to calculate the difference between final and initial position so you need to set the initial position of local odometry to correspond with the initial position of global odometry. Calculate the difference between robot positions so the coordinate system shift will not affect the outcome.

Let θ be an angle between local and global odometry in the initial position.

$$\theta = \varphi_{loc\_start} - \varphi_{glob\_start}$$

You can use rotation matrix to transfer coordinates from the system 1 (local odometry) to the system 0 (global odometry):

$$\begin{pmatrix} x_0 \\ y_0 \end{pmatrix} = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \begin{pmatrix} x_1 \\ y_1 \end{pmatrix}$$

therefore

$$\begin{aligned} x_0 &= x_1\cos\theta - y_1\sin\theta \\ y_0 &= x_1\sin\theta + y_1\cos\theta \end{aligned}$$

3.3.3 Experiment

As mentioned above, it is not posibble to test this program in a simulator. The robot in SyRoTek was set to make 2 squares with the lenght of an edge 0.6 meters. You can see robot's trajectory in the Fig. 3.3.

The difference between the local and the global odometries is as follows:

```
DIFF_X = 0.049710
DIFF_Y = 0.055682
DIFF_ANGLE = -0.192555
```

The difference between the final and the initial positions in global odometry is as follows:

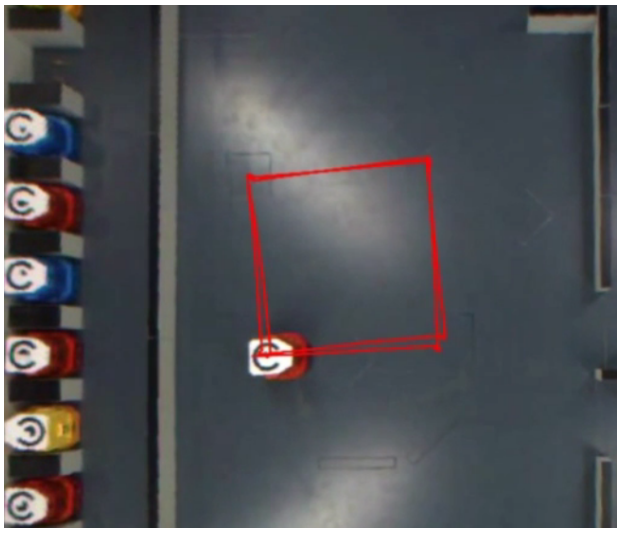


Figure 3.3: The PID controller in SyRoTek

```
DIFF_X = -0.004052
DIFF_Y = -0.046150
DIFF_ANGLE = 0.100000
```

As you can see from the results, localization using local odometry was by ~ 5 centimeters (in each axis) and ~ -0.2 radians different from global odometry.

The video record of this experiment can be found on the SyRoTek website[8].

3.4 Wall Following

The aim of this tutorial is to implement an algorithm that drives the robot along the wall. The robot should keep the same distance from the wall. The first problem is to find, where is the wall. You can use laser rangefinder and find minimal values (as in Braitenberg vehicle tutorial). This way you have the distance from the closest obstacle and the angle relative to the robot.

To succesfully follow the wall, you need to keep the same distance and angle ($\pi/2$ or $-\pi/2$ depending on position of the wall - the left or right side of the robot). You can use two PID like controllers (which will change angular velocity) to maintain these values. Wall following is implemented in the `NodeWallFollowing` class.

[8] https://syrotek.felk.cvut.cz/about/videos

3.4.1 Processing input and output

Input to this algorithm are the data from laser rangefinder (Section 3.1.1) and output is `Twist` message (Section 3.1.1). You need to know the angle of minimal value from the `ranges` vector in laser data. Use the equation 3.1 to calculate it.

If the minimal value was determined from the whole `ranges` vector, the robot could start following the oposite wall in tight spaces. To ensure this does not happen, find minimum only from the appropriate half of `ranges` vector. Finding the minimal value is done in a loop so you can adjust its start and end according to these equations:

$$\begin{aligned} i_0 &= l \cdot \frac{d+1}{4} \\ i_n &= l \cdot \frac{d+3}{4}, \end{aligned}$$

where

d is direction (1 for following wall on left side of robot, -1 for wall on right side),

i_0 is an index of the element in the vector, where the loop starts,

i_n is an index of the element in the vector, where the loop ends and

l is a lenght of the the vector.

3.4.2 PD controller of the direction

You need to calculate error value first:

$$e(t_n) = r_{min}(t_n) - r_{wall},$$

where

$e(t_n)$ is error value,

$r_{min}(t_n)$ is the distance of the closest obstacle (minimal value from the `ranges` vector) measured from laser scan position and

r_{wall} is the desired distance from the wall (reference value).

You can use discrete PD controller to keep desired distance from the wall. The following equation is based on equation 3.4, but only proportional and derivative part is used.

$$\omega_A(t_n) = k_P e(t_n) + k_D \frac{e(t_n) - e(t_{n-1})}{t_n - t_{n-1}},$$

where

$\omega_A(t_n)$ is intervention from PD controller,

t_n is time of n-th iteration,

k_P is proporcional constant and

k_D is derivative constant

To simplify calculation you can assume, that $t_n - t_{n-1}$ is a constant value (in reality it is not, but small diferences will not have significant effect on the controller).

$$\omega_A(t_n) = k_P e(t_n) + \tilde{k}_D(e(t_n) - e(t_{n-1})),$$

where $\tilde{k_D}$ is equal to $\frac{k_D}{t_n - t_{n-1}}$.

3.4.3 P controller of the angle

For controlling the angle of the closest wall suffices a simple proportional controller. Calculate error value first:

$$e_\varphi(t_n) = \varphi_{min}(t_n) - d \cdot \frac{\pi}{2},$$

where $e_\varphi(t_n)$ is the error value, $\varphi_{min}(t_n)$ is the angle of the closest wall and d is a direction (1 for following wall on left side of robot, -1 for wall on right side). Now you can use simple P controller to calculate the intervention:

$$\omega_B(t_n) = k_{P2} e_\varphi(t_n),$$

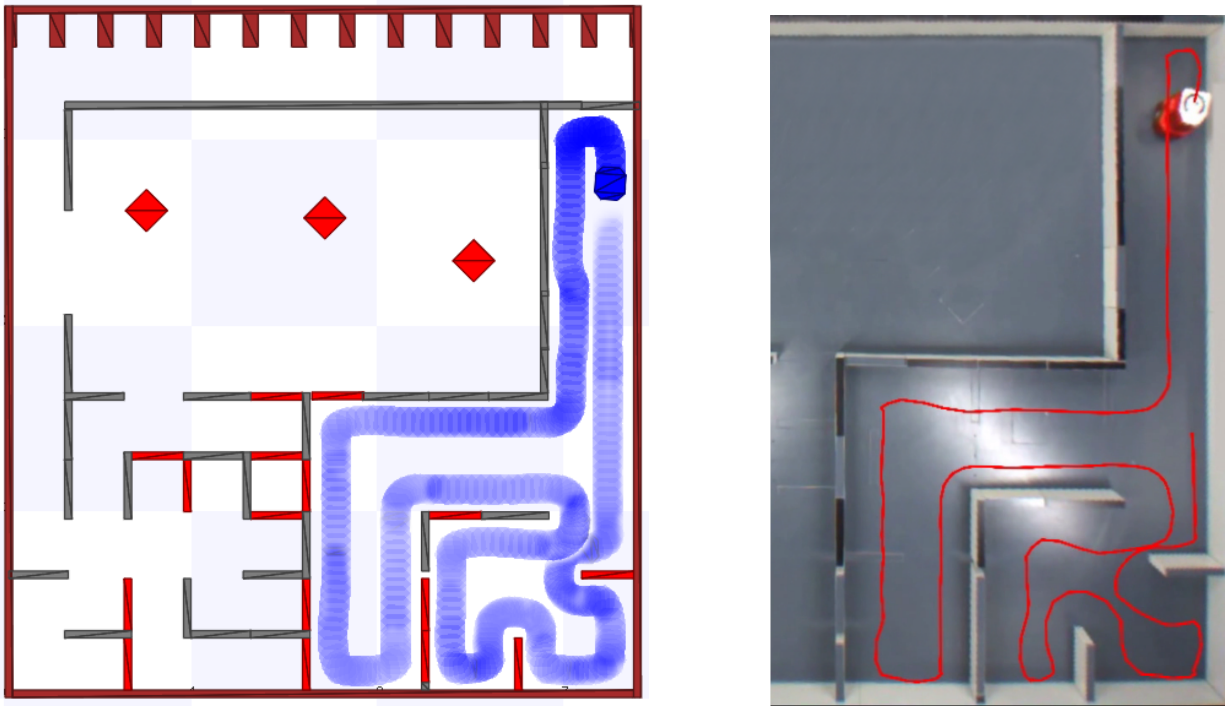
where $\omega_B(t_n)$ is intervention from PD controller, t_n is time of n-th iteration and k_{P2} is proporcional constant. The final angular velocity could be calculated as a sum of interventions from PD and P controller:

$$\omega(t_n) = \omega_A(t_n) + \omega_B(t_n)$$

3.4.4 Linear velocity

If the linear velocity would be constant, robot would not manage to turn in corners in time or the velocity would have to be too low. This could be solved by checking the distance of a point in front of the robot and if it is too close, robot slows down (or stops linear movement completely). Than it has enought time to turn around.

Another issue may occur when the wall ends. The robot might not be able to turn around in time and it could go forward and start following another wall. You can solve it by checking angle of the closest obstacle and if the absolute value of that angle is bigger than 1.75 (which means the obstacle is already the behind the robot), lower the linear velocity.



(a) The experiment in stage simulator

(b) The experiment in SyRoTek

Figure 3.4: The wall following

3.4.5 Experiments

In Fig. 3.4a you can see that the robot is succesfully following a wall in a simulator and it is keeping the same distance. The same experiment as in simulator was done in SyRoTek. As you can see in Fig. 3.4b, precision is lower than in simulator, but the robot still managed to succesfully follow the wall. The real precision is higher than in the image, because image is slightly deformed from camera. The robot did not hit or loose the wall. The video record of this experiment can be found on the SyRoTek website[9].

3.5 Trajectory Following - Pure Pursuit

The aim of this task is to show how to write a pure pursuit algorithm in ROS, which is the algorithm for following the predetermined trajectory. A control target point is generated on the trajectory (in a constant distance from the robot), which represents an actual target. The robot calculates a circular trajectory, which crosses the control point and an actual rotatation of the robot is tangent to that trajectory.[16]

[9] https://syrotek.felk.cvut.cz/about/videos

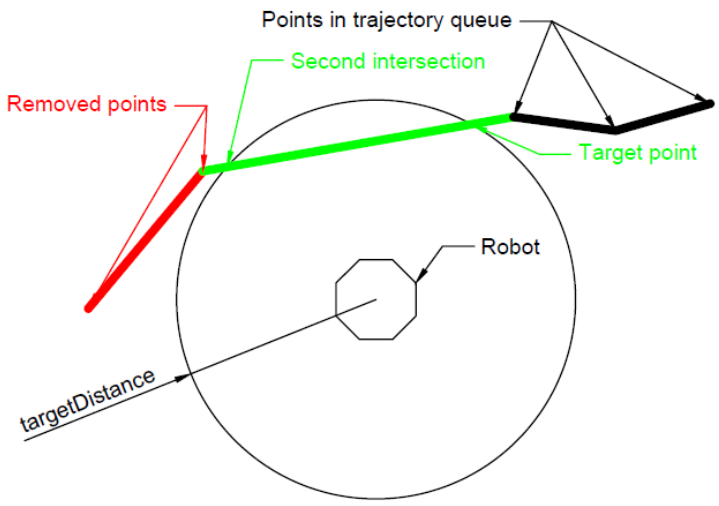


Figure 3.5: Target point on the trajectory

Pure pursuit algorithm is implemented in the `NodeTrajectoryFollowing2` class. To make calculation easier, create a class for positions called `MyPoint` (you can use the one created in PID controller, but it needs to add few methods). The input trajectory is stored as a queue.

Commands for the robot are sent through the `Twist` messages. You can find more information about this type of message in Section 3.1.1. Input of this program are the `Odometry` messages. To see the detailed description look at the Section 3.2.1.

3.5.1 Positions and vectors

As was mentioned before, there is a separate class `MyPoint` to store informations about positions. This class behaves as a 2D vector (it has x and y variables and methods for vector operations) and has several new methods (apart from those in MyPoint class from PID controller). These methods represent operations like norm, multiplication by real number, adding and substracting. They are used mainly for easier work with vectors in the calculation of the control target point (Section 3.5.3).

3.5.2 Trajectory

The trajectory is stored as a queue of objects (`MyPoint`). First (before the movement starts, in the constructor) remove a front point from the trajectory and save it to the new variable (`lastRemoved`). After that you need to start removing points from the trajectory, which are already considered as achieved (they are closer to the robot than the predefined constant distance of the target control point). This step ensures that the robot does not start following the trajectory backwards.

Always keep the last removed point in the `lastRemoved` variable. If none point (except for the one removed in the constructor) was removed, the robot goes first to the point removed in the constructor. Otherwise the algorithm finds the control point on the trajectory (between the `lastRemoved` and the front one in the trajectory). For the better understanding look at Fig. 3.5.

3.5.3 Control target point

As you can see from Fig. 3.5, the control point lies on the intersection between the line (defined by two points - the last one removed and front one in the trajectory queue) and the circle (with a center in the actual position of robot and the radius equal to the `targetDistance`). To avoid calculation of line-circle intersection (which would provide two intersections and one of them would have to be chosen), you can approximate the position of the target point.

$$d = ||\vec{l} + \vec{s} \cdot 2^{-i} + \vec{v} - \vec{a}||,$$

where

$\vec{l}$ is the last removed point from trajectory,

$\vec{s}$ is equal to $\vec{f} - \vec{l}$ ($\vec{f}$ is the front point in the trajectory queue),

$\vec{v}$ is contribution of previous iterations,

$\vec{a}$ is the actual position of the robot,

$\vec{d}$ is a distance between the actual position of the robot and the estimated position of the target point and

i is the number of the current iteration.

If d is smaller than `targetDistance`, add $\vec{s}$ to $\vec{v}$ before the next iteration:

$$\vec{v}_{i+1} = \vec{v}_i + \vec{s}_i$$

After several iterations (~ 10) you have an approximated position of the target point with sufficient accuracy ($\sim 1/2^{10} \cdot ||\vec{s}||$):

$$\vec{t} = \vec{l} + \vec{v}$$

If the robot would be so far from the trajectory, that it would not remove more points than the first one, which is further from the robot than `targetDistance`, vector $\vec{v}$ would remain $\vec{0}$ and robot would go to the first removed point.

If the trajectory would have its points far from each other and the robot would be positioned as in Fig. 3.6, the algorithm explained above might find a second intersection, which would

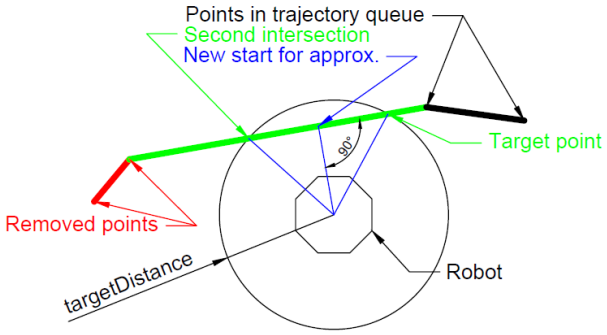


Figure 3.6: Two intersections

cause the robot to go backwards. The robot always needs to go to the itersection closer to the front point in the trajectory so set up the vector $\vec{v}$ before approximation so that the algorithm starts approximating from the point between these two intersections (the unwanted one will be skipped). The easiest way to do that is to find an intersection between the trajectory and the line perpendicular to the trajectory passing through $\vec{a}$.

$$\begin{aligned} x_v &= \frac{||\vec{a}-\vec{l}|| \cdot \cos(\varphi_{\vec{f}-\vec{l}} - \varphi_{\vec{a}-\vec{l}})}{||\vec{s}|| \cdot x_s} \\ y_v &= \frac{||\vec{a}-\vec{l}|| \cdot \cos(\varphi_{\vec{f}-\vec{l}} - \varphi_{\vec{a}-\vec{l}})}{||\vec{s}|| \cdot y_s}, \end{aligned}$$

where $\varphi_{\vec{f}-\vec{l}}$ is the angle between $\vec{f}$ and $\vec{l}$ and $\varphi_{\vec{a}-\vec{l}}$ is the angle between $\vec{a}$ and $\vec{l}$.

3.5.4 Circular trajectory

Now you need to calculate the circular trajectory, which crosses the target point and the actual rotatation of the robot is tangent to that trajectory. First you need to get the angle between the vector from the actual position to the target point and the direction of the robot (φ in Fig. 3.7).

$$\varphi = \varphi_{\vec{t}-\vec{a}} - \alpha,$$

where $\varphi_{\vec{t}-\vec{a}}$is the angle between the target and the actual position.

Next you need to know the radius of the circle on which the new trajectory lies.

$$r_{new} = |\frac{||\vec{t}-\vec{a}||}{2 \cdot \cos(\frac{\pi}{2} - \varphi)}|$$

Now set up the angular velocity (the linear velocity is constant) to follow the calculated trajectory. Find the distance between the actual and the target position (on a circular trajectory).

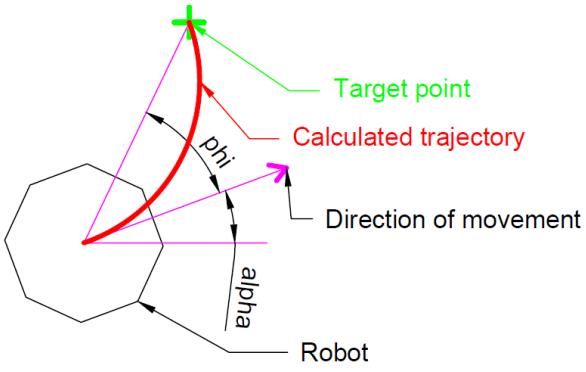


Figure 3.7: Circular trajectory

To do that, you need to know the angle of the circular section, which will be 2φ(at the end should be the robot's direction again tangent to the circular trajectory so it needs to rotate the angle φ twice).

$$d_{new} = 2\pi r_{new}\left(\frac{|2\varphi|}{2\pi}\right) = 2\pi r_{new}\left(\frac{|\varphi|}{\pi}\right)$$

The linear velocity is constant so you can simply get the time that the robot will need to go through the entire circular trajectory to the target point.

$$t = \frac{d_{new}}{v},$$

where t is time and v is linear velocity.

The robot needs to rotate twice the angle φ in time t. That gives you the angular velocity:

$$\omega = \frac{2\varphi}{t}$$

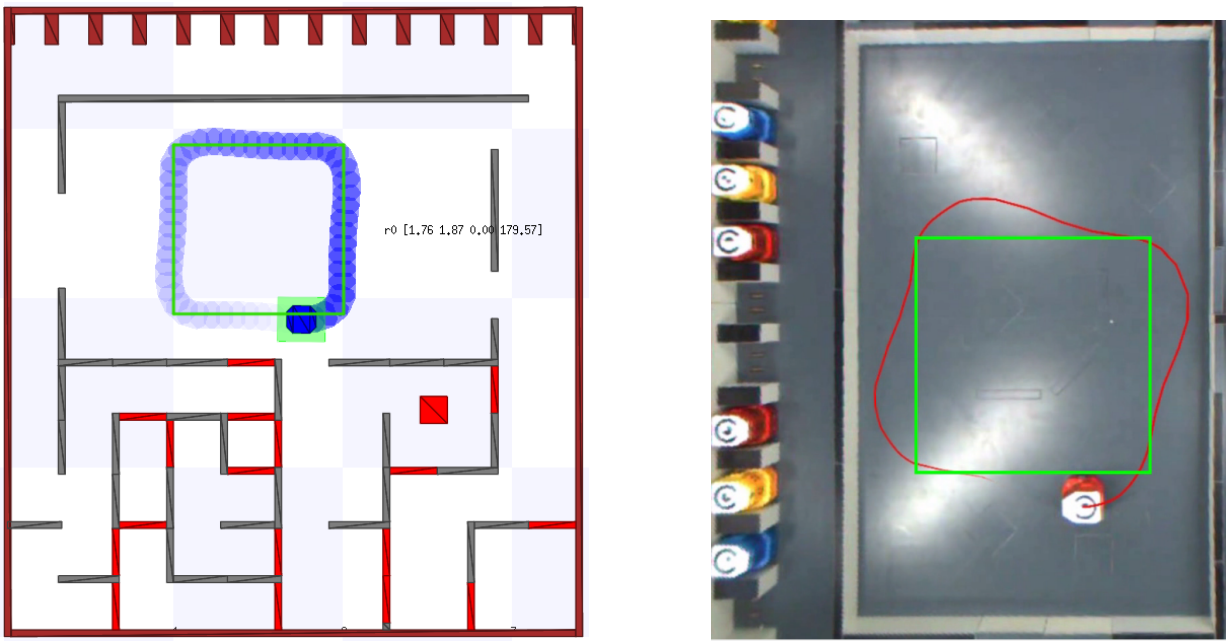
The linear and angular velocities can now be sent through the `Twist` message to the robot.

3.5.5 Experiments

The program was tested on a repeating square trajectory ($[1.0; 1.9]$, $[1.0; 2.9]$, $[2.0; 2.9]$, $[2.0; 1.9]$) which is marked as a green square in Fig. 3.8a.

As you can see from Figs. 3.8a and 3.8b, deviation from the predefined trajectory is much bigger on SyRoTek than in a simulator. This might happen, because the real robot is not able to turn as quickly as the robot in a simulator while maintaing constant linear speed and the real angular velocity of robot is smaller than the one sent through `Twist` message. The video record of this experiment can be found on the SyRoTek website[10].

[10] https://syrotek.felk.cvut.cz/about/videos



(a) The experiment in the Stage simulator

(b) The experiment in the SyRoTek

Figure 3.8: The pure pursuit

Chapter 4

Parameter server and launch files

In large projects you need to run several nodes and set up many parameters. In order to simplify work with large projects, the ROS provides parameter server and launch files. The parameter server provides a way to set up the variables (even in runtime) without parsing arguments and the launch files are used for starting large projects with several nodes through one file in `xml` format.

4.1 ROS Parameter Server

According to the authors of ROS, the parameter server is a shared, multi-variate dictionary that is accessible via network APIs. Nodes use this server to store and retrieve parameters at runtime. As it is not designed for high-performance, it is best used for static, non-binary data such as configuration parameters. It is meant to be globally viewable so that tools can easily inspect the configuration state of the system and modify if necessary [17].

The parameters can be of different types (integers, booleans, doubles, strings, ...). They have the hierarchy corresponding with the names of nodes and topics, so the parameters of the node named `node1` would be: `/node1/parameter1`, `/node1/parameter2`, ...

To get private parameters (parameters that belong to the node, from which you are accessing them), you can also use tilde ~ instead of the name of a node. Accessing `/node1/parameter1` from `node1` would be simply through `~parameter1`. To set the parameter, you can call `rosparam set` in the command line or you can use remapping arguments when starting node from the command line (this time the tilde ~ would by replaced by underscore _) [17].

```
rosrun <package> <executable> _parameter1:=1.0 _parameter2:=true
```

Only local parameters are used in this thesis. To access them in C++ program, it is best to start `NodeHandle` in tilde ~ namespace.

```
ros::NodeHandle n("~");
```

To access parameteres you can use the function `ros::NodeHandle::getParam` [18].

```
double tol;
if (!n.getParam("tolerance", tol))
    tol = 0.05;
```

This is an example of reading parameter from the server. Function `getParam()` saves a value from the `tolerance` parameter to the `tol` variable. If this fails, the default value (0.05) is used. The parameters could also be set or deleted from the C++ program.

4.2 Roslaunch

If you need to run multiple applications in the ROS, starting a new command line for each of them might be confusing. Fortunately ROS provides a roslaunch package to run multiple applications and set up their parameters through one single launch file (in XML format). It can also read `yaml` files with parameters. The `roscore` is actually specialized a roslaunch that brings up all the core parts of the ROS system.[19]

You can launch the XML configuration file by using the command:

```
roslaunch file_name.launch
```

To see a detailed description how to write launch files, look at [19]. The simple launch file could look like this:

```
<launch>
  <node name="how_to_name_node" package="package_name" type="
   executable_name" args="arguments">
    <param name="parameter1" value="1.0"/>
    <rosparam file="parameters.yaml" command="load"/>
  </node>
</launch>
```

The parameters for a node could be set by three ways. The first one is to write them into arguments in the way described in Section 4.1. The second one is to write them by using `<param>` tag. The third one is recommended for a large number of parameters and it consists in including `yaml` file. The parameters in the yaml file would be written like this:

```
parameter1: 0.1
```

The work with parameters is very simplifed if you use launch file together with the parameter server. You do not need to write parameters into a command line or to compile a program each time when some parameter changes. You can simply set up all parameters in the launch file.

Chapter 5

Transformations

Because ROS is programmed as a universal system, it needs to work on more types of robots than a simple mobile robot (as used in the SyRoTek). If some robot has one or more joints, it also has more than one coordinate frames. It is possible to keep track of these frames by using multiple odometry topics, but it would not be easy for a programmer to get for example transformation between last two joints of a robot. To simplify work with these transformations, ROS provides the `tf` package. It is a package, that maintains relationship between particular coordinate frames in a tree structure [20].

Even though robots in SyRoTek do not have any joints, transformations might be still usable (for example transformation between robot's position and a laser rangefinder or between a map and robot's position).

5.1 How transformations work

To see a simple example of a transformation, start the Stage simulator in ROS (with a model of SyRoTek arena) and use the command:

```
rosrun tf view_frames
```

This command generates a pdf file which shows information about actual transformations in a graph (figure 5.1). The Stage simulator provides three transformations. The first one is between the `/odom` (the odometry coordinate frame is fixed) and the `/base_footprint` frame. This transformation should correspond with the position of the robot transmitted through the `/odom` topic. The second transformation is in this case equal to the vector of zeros. The third transformation is however very useful, because it describes the position of the laser scanner (which is in this case positioned 4 cm forward from the center of robot).

The main benefit is that ROS automatically calculates all reverse transformations and even transformations through several steps. For example the one from the `/odom` frame to the `/base_laser_link` frame.

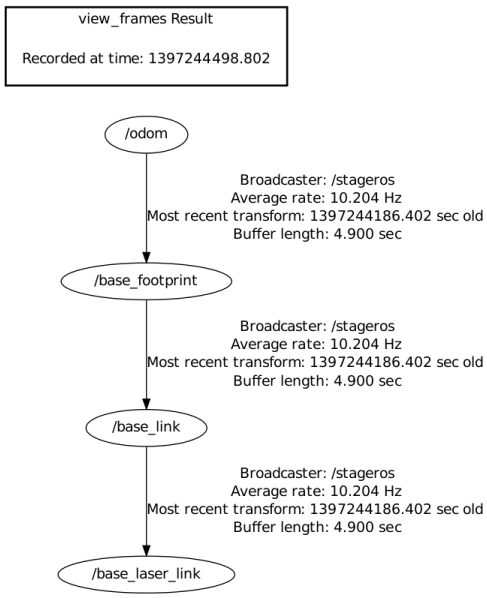


Figure 5.1: Transformation graph from the Stage simulator

To look at some transformation use the command [20]:

```
rosrun tf echo <source_frame> <target_frame>
```

For example:

```
rosrun tf echo /odom /base_laser_link
```

The result is:

```
At time 1395.200
- Translation: [1.240, 2.000, 0.150]
- Rotation: in Quaternion [0.000, 0.000, 0.000, 1.000]
            in RPY [0.000, -0.000, 0.000]
```

The transformation is in a similar format as odometry messages. It contains a translation vector and a quaternion representing rotation. It also provides rotation in the "roll-pitch-yaw" format. To create new static transformation you can simply use the command [20] :

```
rosrun tf static_transform_publisher x y z yaw pitch roll <frame_id> <
   child_frame_id> <period_in_ms>
```

or

```
rosrun tf static_transform_publisher x y z qx qy qz qw <frame_id> <
   child_frame_id> <period_in_ms>
```

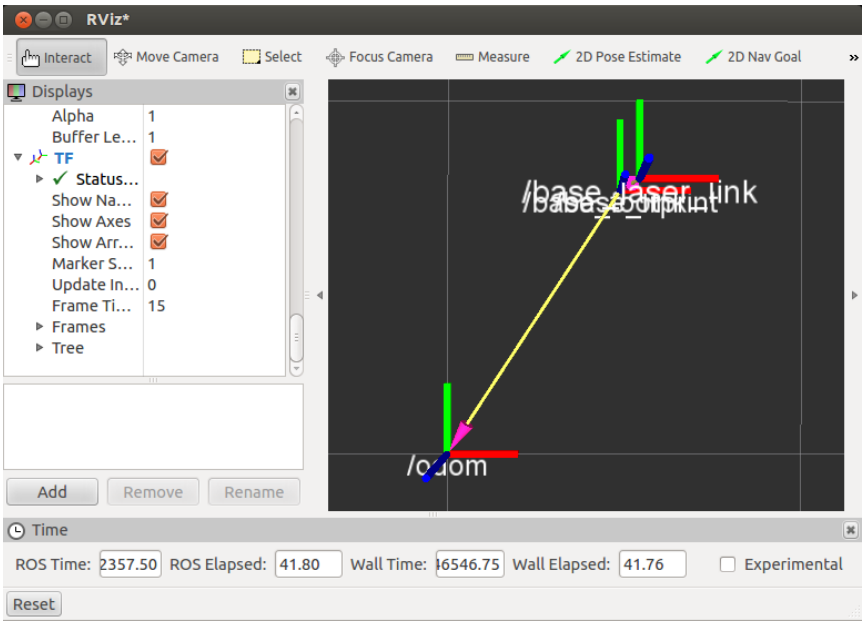


Figure 5.2: Transformations in RVIZ

5.2 Viewing transformations in RVIZ

The transformations could be viewed by using RVIZ. You simply add TF, set the fixed frame (in this case `/odom`) and RVIZ automatically visualizes all coordinate frames and updates them in real time (Fig. 5.2).

5.3 Using the SyRoTek transformation tree

The robot in SyRoTek uses two localization systems: the global one (external localization by the camera on top) and the local one (measured from movement of wheels - less precise, but its behaviour is closer to the robot in open space without external localization). The `/arena` frame is used as fixed instead of `/odom` as in a simulator. For each odometry is one frame corresponding with `/base_footprint` in a simulator. These frames are `/syros/global_odom` and `/syros/base_odom` (see Fig. 5.3). Frame `/syros` should correspond with `/base_link` and `/syros/laser_ranger` with `/base_laser_link`.

Usage of transformations in global odometry is similar as in a simulator (it is only necessary to use corresponding frames). Using local (base) odometry might be more problematic, because the frame of the laser scanner is in a global odometry subtree. The easiest way to fix this would be publishing a static transformation (Section 5.1). The laser sensor is positioned 4 cm in front of the center of the robot, therefore the 'x' in a static transformation would be equal to 0.04.

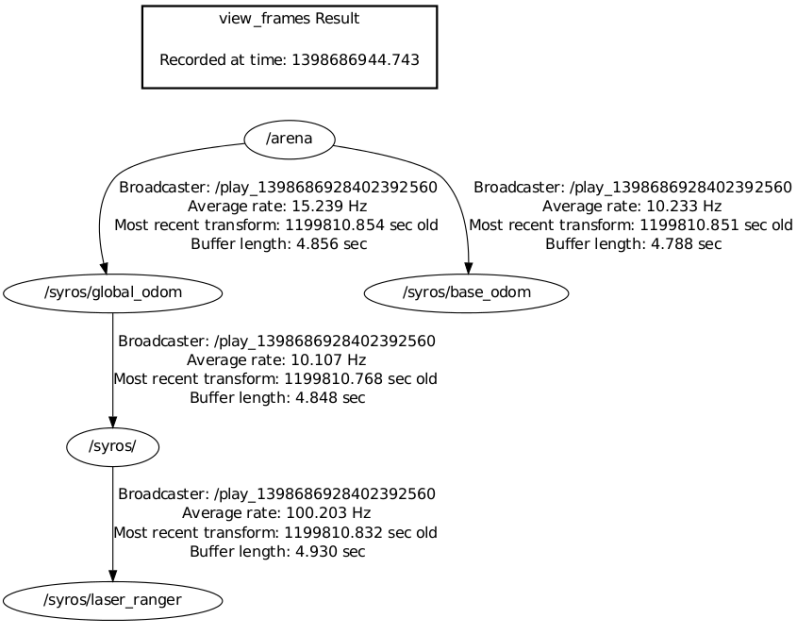


Figure 5.3: Transformations in the SyRoTek

5.4 Transformation timing

Each transformation sent to `/tf` topic has its own time stamp. Each time stamp is generated by the machine, which sends the transformation. This might cause large problems when some nodes are running on another machine than `roscore`. Some applications in ROS (like gmapping, RVIZ, navigation stack) are comparing a time stamp with a time on local machine and some of those applications have very small tolerance. If the time on the local machine is not synchronized with the global machine, applications ussualy discard all transformations.

The last possible way to solve this issue (if synchronization fails and tolerance can not be adjusted) is to resend all transformations from the local machine with its own time stamp.

Chapter 6

Exploration with known pose

The goal of the exploration task is to create a map of an unknown environment. It contains three separate programs: mapping (for creating map), planning (for finding a path in that map) and following (for following the found path). Each one of them is independent and can be used on its own. It is assumed that the position and rotation of the robot in this task is known (ideal localization). Even though the external localization of the SyRoTek is not ideal, it is sufficient for this task.

Originally all three exploration tasks were written to work only with external localization through global odometry and did not use transformation or the parameter server. Later it was necessary to use the planning part in exploration with its own localization (see Chapter 7). It was rewritten (as well as the following part) to use transformations and the parameter server.

The process of each iteration in the exploration starts with acquiring the data from the laser rangefinder and the odometry (lines 2-3, Section 6.2.1). These data are processed into the map which is refreshed (lines 4-5, Section 6.2.3). The planning algorithm (lines 6-9, Section 6.3) uses that map for finding the path to the nearest frontier[1]. The path is found through the Djikstra's algorithm (line 7, Section 6.3.6). Finally the path is sent to the following part, which guides robot safely through that path (lines 10-13, Section 6.1).

[1]Frontier is a cell in the map, which is on the edge of an unknown space.

```
1  repeat
2      Input: laser data
3      Input: robot position
4      integrate measurement into the map;
5      Output: refreshed map
6      expand obstacles;
7      find a trajectory to the nearest frontier;
8      clean the trajectory;
9      Output: trajectory
10     repeat
11         find a control target point on the trajectory;
12         go to the control target point;
13     until the robot is at the end of trajectory;
14 until a reachable unknown space does not exist;
```

Scenario 1: The exploration task

6.1 Following - Follow the Carrot

The program uses "Follow the Carrot" algorithm for following the predetermined trajectory. A control target point is generated on the trajectory (at a constant distance from the robot), which represents the actual target. The robot calculates angular difference between its orientation and the direction to the control point and regulates it to zero [16].

This task is based on the pure pursuit algorithm (Section 3.5) so only the different parts are explained. In order to use this algorithm in the exploration task it is necessary to change the trajectory at runtime and some simple obstacle avoiding should be used, because trajectory calculated during the exploration might not be precise and there is a risk that the robot hits some obstacle.

Follow the carrot algorithm is implemented in the `TrajectoryFollow` class. To simplify calculations with vectors and points in 2D space, use `MyNode` class created in the pure pursuit algorithm. The program calculates the position of the control target point (using the same algorithm as in the pure pursuit) and then the angular velocity.

This time however the following does not use data from the odometry messages. To ensure better compatibility with another programs, transformations are used in order to get the position of the robot. First, when new path arrives, it is necessary to convert it from its own frame to the odometry frame (in the case of using mapping from Section 6.2, the frames will be the same). This conversion is necessary, because if the base of the path coordinate frame, which is usually the same as the map coordinate frame, changes drastically, robot's movement would be too shaky.

When this is done, the program waits for the data from the laser scanner (necessary for obstacle avoidance). After these data arrive, transformation from the odometry frame to robot's

frame is used in order to get the position. The data from this transformation should correspond with odometry messages.

Apart from the pure pursuit algorithm, this time a simple obstacle avoidance is implemented.

6.1.1 Processing inputs and outputs

Commands for the robot are sent through the `Twist` messages. You can find more information about this type of message in Section 3.1.1.

Input is from the `LaserScan`, which is necessary for the obstacle avoidance. For more detailed information about this type of message look at the Section 3.1.1. To find the closest obstacle find the minimum value from the `ranges` vector and calculate the appropriate angle of this value by using the equation 3.1.

The third input is the path to be followed. The `Path` message contains the following variables:

```
std_msgs/Header header
geometry_msgs/PoseStamped[] poses
  std_msgs/Header header
  geometry_msgs/Pose pose
    geometry_msgs/Point position
    geometry_msgs/Quaternion orientation
```

In this case, the path only has positions in `x` and `y` coordinates. You can simply create a new instance of `MyPoint` class for each element in the `poses` vector and add it to the trajectory queue. Do not forget to empty the trajectory before adding new points. In this case the trajectory changes every iteration and you need to follow always the newest one. Transform each trajectory to the odometry frame.

6.1.2 Angular velocity

If you have found control target point (the algorithm is exactly the same as in pure pursuit), now you need to calculate the angular velocity. First calculate the angle representing the error value. It is the difference between the direction to control target point and the actual orientation of robot:

$$\varphi = \varphi_{\vec{t}-\vec{a}} - \alpha,$$

where φ is error value, $\varphi_{\vec{t}-\vec{a}}$ is the angle between the target and the actual position and α is robot's orientation.

Now you have the angle. To transfer this angle to angular velocity use the equation:

$$\omega = k \cdot \varphi,$$

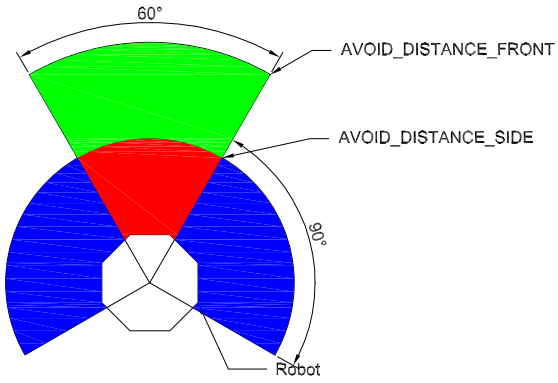


Figure 6.1: Obstacle avoidance

where ω is result angular velocity and k is constant equal to $\frac{1}{t}$ (t - time for returning to zero error value).

Higher value of constant k provides faster reactions on trajectory changes, but it may cause unstable behaviour. Recommended value is $k = 2$.

6.1.3 Simple obstacle avoidance

The Follow the Carrot algorithm uses simple obstacle avoidance. First find the position of the closest obstacle relative to robot. Then, if the obstacle is in the green zone (Fig. 6.1), slow the robot to 80% of original velocity and set error angle to be:

$$\varphi = -0.3 \cdot sign(\beta),$$

where β is angle of closest obstacle.

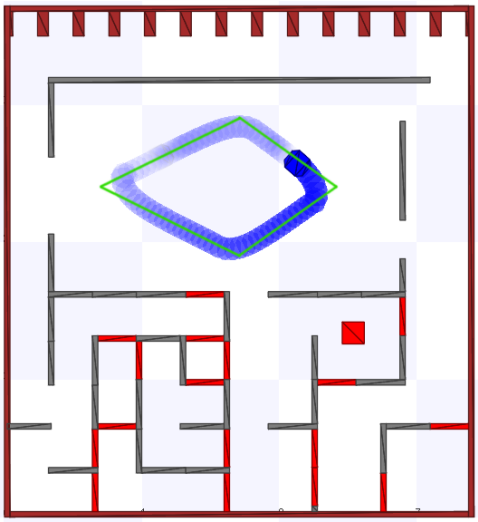
If it is in the blue zone, slow down the robot to 60% of original velocity and set the error angle to be:

$$\varphi = -0.2 \cdot sign(\beta)$$

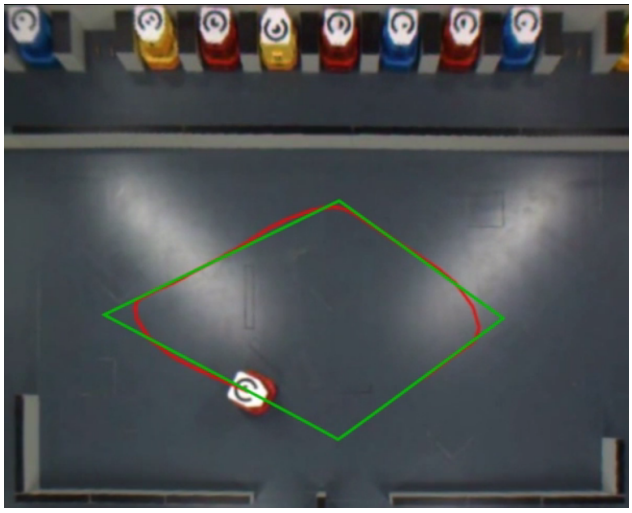
If it is in the red zone, slow down the robot to 60% of original velocity and set the error angle to be the same as in the green zone.

6.1.4 Experiments

The program was tested in Stage simulator with the prepared trajectory (which was submitted through the `/path` topic). The trajectory was: $[1.7; 2.9], [0.7; 2.4], [1.7, 1.9], [2.4, 2.4]$ repeated several times. In Fig. 6.2a you can see that the robot was succesful in following that trajectory.



(a) The experiment in the Stage simulator



(b) The experiment in the SyRoTek

Figure 6.2: Follow the carrot

Program was also tested in the SyRoTek. The same trajectory as in a simulator was used and you can see the result of this test in Fig. 6.2b.

The real trajectory of the robot is very similar to the simulated one. That did not happen in the pure pursuit algorithm, because the robot was not slowed down there in sharp turns and the behaviour of the simulator and the real robot is very different, when robot needs to rotate fast while maintaining constant linear velocity.

6.2 Mapping

The mapping part of the exploration task creates a map of the unknown environment from laser data. The main part of this algorithm is in the class named `Grid`. It contains an occupancy grid[2] and methods for input and output. This program has two inputs: odometry, which contains information about the position of the robot and the laser scan. Each measured data from the laser scan are processed into the occupancy grid and the final map is sent through the `/map` topic.

6.2.1 Processing inputs

First input of this program are the `Odometry` messages. Processed data from odometry are saved to the appropriate variables in the `Grid` class . Before doing that, you need to adjust x

[2]Occupancy grid is a grid of cells, where each cell has a value representing the probability of an obstacle on the position of that cell.

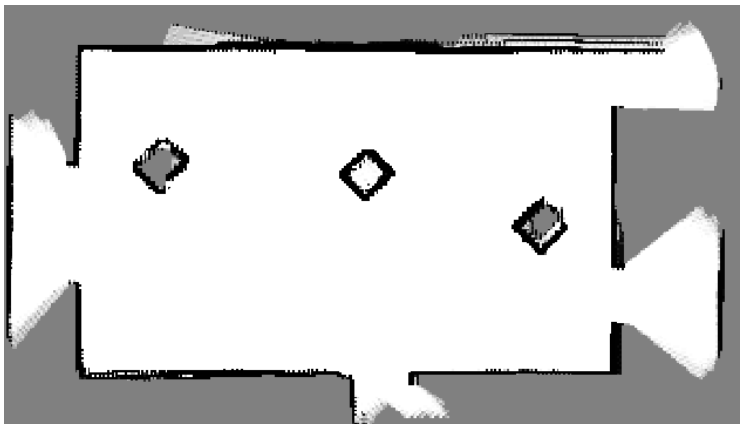


Figure 6.3: Occupancy Grid

and y position, because the sensor is positioned $4cm$ forward from the center of robot:

$$\begin{aligned} x_{new} &= x + 0.04 \cdot \cos(\varphi) \\ y_{new} &= y + 0.04 \cdot \sin(\varphi) \end{aligned}$$

Another input is from the laser scan. Data from the laser scan are used (together with the odometry data) to create a map.

6.2.2 Occupancy grid

The main goal of using the occupancy grid is to represent the environment by a grid and estimate the probability that a location is occupied by an obstacle (look at Fig. 6.3). To use the occupancy grid you need two assumptions: occupancies of individual cells are independent and the robot's position is known [21]. In this case, you can store the occupancy grid as `std::vector<double>`. Size of this vector is equal to `size_x*size_y`, where `size_x` is a number of cells in x axis and `size_y` in y axis This vector represents a 2D grid so you need to make representation of a cell in the two-dimensional grid to the one dimensional array:

$$k = j \cdot l_x + i, \tag{6.1}$$

where $[i, j]$ are coordinates of target cell, k is position in the vector (or 1D array) and l_x is `size_x`.

Updating a value in the specified cell can be based on the Bayes rule:

$$p(A|B) = \frac{p(B|A) \cdot p(A)}{p(B)}$$

This equation can be modified for usage in the occupancy grid:

$$occ(i,j)_n = \frac{occ(i,j)_{n-1} \cdot p(O)}{occ(i,j)_{n-1} \cdot p(O) + (1 - occ(i,j)_{n-1}) \cdot p(F)}, \tag{6.2}$$

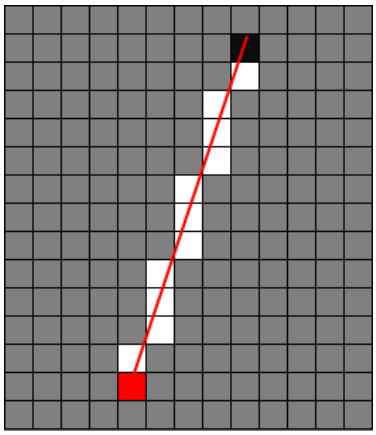


Figure 6.4: One measure from the laser.

where:

$occ(i,j)_n$ is probability that the cell (i,j) is occupied,

$occ(i,j)_{n-1}$ is probability that the cell (i,j) is occupied according to the last iteration,

$p(O)$ is probability that the cell (i,j) is occupied according to the last measurement (0.85 if this cell was evaluated as occupied in the last measurement, 0.35 if this cell was considered as empty in the last measurement) and

$p(F)$ is probability that the cell (i,j) is free according to the last measurement ($p(F) = 1 - p(O)$).

6.2.3 Processing measurement

Each value in the laser scan data is processed separately. If you focus on one value, you have:

- the distance from the sensor according to k-th measurement: d_k
- the angle relative to the sensor angle: φ_k
- the position and orientation of the sensor from odometry: $[x, y, \varphi_R]$
- lenght of an edge of the cell: s

Now transfer x and y coordinates for the position of sensor to the coordinates in the grid:

$$x_g = \left[\frac{x}{s}\right]$$
$$y_g = \left[\frac{y}{x}\right]$$

Calculate the position of the obstacle in the grid:

$$\begin{aligned} x_{ok} &= x_g + \frac{d_k}{s} \cdot \cos(\varphi) \\ y_{ok} &= y_g + \frac{d_k}{s} \cdot \sin(\varphi) \end{aligned}$$

and again round those values. Start in the $[x_g, y_g]$ coordinates and in each iteration use the equation 6.2 with $p(O) = 0.35$ and $p(F) = 0.65$ for the cell:

$$\begin{aligned} x_{cell} &= x_g + i \cdot \cos(\varphi_k) \\ y_{cell} &= y_g + i \cdot \sin(\varphi_k), \end{aligned}$$

where i means the i-th iteration.

Skip the cells, that would be evaluated twice. That might happen (if $\varphi \neq k \cdot \frac{\pi}{2}; k \in \mathbf{Z}$).

$$d_g = \sqrt{(x_{ok} - x_g)^2 + (y_{ok} - y_g)^2}$$

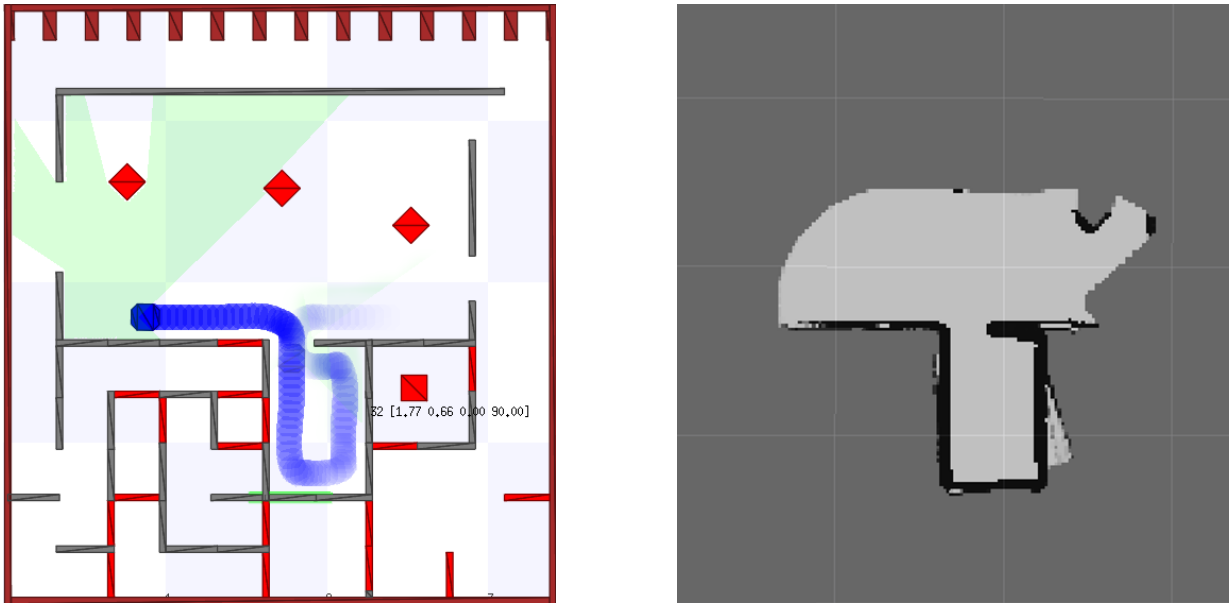
If $j \geq d_g$ or $x_{cell} = x_{ok}$ and $y_{cell} = y_{ok}$, stop the cycle and use the equation 6.2 for $[x_{ok}, y_{ok}]$ with $p(O) = 0.85$ and $p(F) = 0.15$. This means, that all cells until this one are considered as free (in this iteration) and this cell $[x_{ok}, y_{ok}]$ is considered as an obstacle.

If j in the cycle would be bigger than is the reliable distance from sensor ($\sim 0.7m$ converted to cells), stop the cycle and do not save value for the obstacle. You can see the behaviour of this algorithm in Fig. 6.4. The red line is connecting the position of the robot (red cell $[x_g, y_g]$) with the position of the obstacle (black cell $[x_{ok}, y_{ok}]$). The length of the red line is equal to the measured value. The white cells are cells from each iteration considered to be empty $[x_{cell}, y_{cell}]$.

6.2.4 Preparing output

Output of the mapping part of the exploration task will be `OccupancyGrid`:

```
std_msgs/Header header
nav_msgs/MapMetaData info
  time map_load_time
  float32 resolution
  uint32 width
  uint32 height
  geometry_msgs/Pose origin
    geometry_msgs/Point position
    geometry_msgs/Quaternion orientation
int8[] data
```



(a) The path in the simulator

(b) The created map

Figure 6.5: Mapping in the simulator

This message must be prepared with all variables, if you want to be able to use another applications (like RVIZ). For the header and `map_load_time` in info you can use the data from the message with laser scan. It might be good to have same `seq` number in both messages (you can see immediately, which belongs to which). Set the `frame_id` to `odom` in Stageros or `/syros/global_odom` in SyRoTek, because global odometry uses the same coordinate system as `OccupancyGrid`.

The `resolution` variable should be equal to the size of one cell in meters and the `width` and the `height` are number of cells in `x` and `y` direction. The origin is in the point $[0; 0]$ with rotation 0 radians so fill it with zeros. The actual values from the grid are saved in the `data` variable (in percents) as a `signed char`. To transfer the values from the saved grid (which is in $< 0; 1 >$) multiply it by 100 and convert it to a `signed char`.

6.2.5 Experiments

The mapping part of the exploration task does not control robot's movement. For that use the planning (which will be in the next Section) and the following (from the previous Section). To check, if everything is working correctly, another program for controlling the robot's movement was used (wall following).

The map was observed by RVIZ. To see the map there add a Map with `/map` topic.

As you can see in the Fig. 6.5, the program succesfully created the map of the environment and the occupancy grid transmitted through `/map` topic was displayed in rviz (the right image

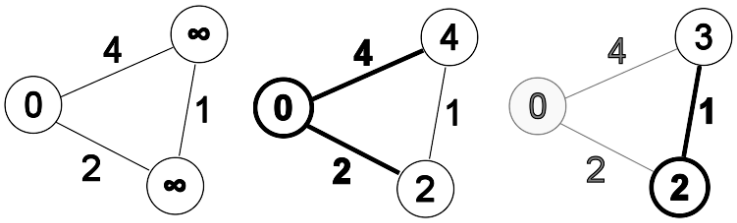


Figure 6.6: Demonstration of Dijkstra's algorithm on a small graph, showing two relaxation operations [1]

in Fig. 6.5).

6.3 Planning

The goal of this task is to find the path to the nearest frontier (which is the cell between a free and unknown space). This program uses the map from the mapping part in order to make a plan. Dijkstra's algorithm (Section 6.3.1) on a connection graph of grid cells is used.

The planning algorithm is implemented in the `Planner` class. Input to this application is the `OccupancyGrid` from the mapping (Section 6.2). First you need to process the input data. The robot is considered as a point in planning, therefore the obstacles have to be enlarged. This is done by expanding obstacles to the free space in the map. Next comes the main part of planning, which is the Djikstra's algorithm.

Expanding obstacles on the whole map before the start of planning was very computationaly demanding, mainly in cases, when the map was much larger than the arena[3]. The program was modified to add new nodes to the graph representation of the map (Section 6.3.3) and expand obstacles around them simultaneously with the actual planning. This modification significantly lowered demands on the computer performance in cases, when the closest frontier is not far from the robot.

The Dijkstra's algorithm finds a path from the robot to the target point (in this case the closest reachable frontier). If the path was found, you need to clean it (remove nodes, which lies on the line between the other two nodes). Finally you can convert the path to the `nav_msgs::Path` (see the Section 6.1.1) and publish it.

6.3.1 Dijkstra's algorithm theoretically

Djikstra's algorithm is a graph search algorithm that finds the shortest path in a graph with non-negative edge path costs. First it expands the start node and marks expanded nodes as "opened". Next it finds an opened node with the lowest cost from the start, expands it and marks it as "closed", which means, it can not be expanded again. If the target is found, the algorithm can trace back the shortest route. It is possible to find a way to all frontiers, but it is not necessary for this program [22].

6.3.2 Processing inputs

The input is `OccupancyGrid` from the mapping part. For detailed description look at the Section 6.2.4. The size of the map (number of cells in a row and column) can be found in the variables `info.width` and `info.height` which are stored in the `OccupancyGrid` message. The size of one cell is stored in the variable `info.resolution`. The `data` vector (containing occupancy grid) is one-dimensional, but it represents a two-dimensional grid so you need to make a representation of a cell in a two-dimensional grid to a one-dimensional array. Use the equation 6.1. Keep the origin of the map[4], because it is necessary for calculating the position of the robot in the map and publishing the path.

To get the position of therobot, look up transform between the map and the robot frame. Convert that position into cell coordinates. This is the moment, when the origin of the map is used. Transform the robot position according to the origin:

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} \cos(\varphi) & -\sin(\varphi) \\ \sin(\varphi) & \cos(\varphi) \end{pmatrix}^{-1} \begin{pmatrix} x_{robot} - x_{origin} \\ y_{robot} - y_{origin} \end{pmatrix} \cdot \frac{1}{s}, \tag{6.3}$$

where φ is the rotation of the origin and s is the size of one cell.

6.3.3 Graph

In order to use the Djikstra's algorithm you need to have a graph representation of the occupancy grid. The representation is made by a node structure (for a cell in the map) and `unordered_map`[5] which contains all nodes. The key in the map is a location of the node converted to 1D coordinates (6.1). Each cell has connection to its neighbours according to the Fig. 6.7. Longer connections (the green lines) should have the cost equal to $\sqrt{2}$ of shorter

[3] If you use some external mapping (for example Gmapping - Section 7.2), it might automatically expand the map far beyond borders of the arena. This situation does not occur when you use mapping from previous section.

[4] The origin from mapping is equal to $\vec{0}$. It might seem unnecessary to work with it, but it ensures compatibility with another packages (like gmapping 7.2).

[5] The `unordered_map` is a C++ structure representing the map and allowing to access the elements by a key.

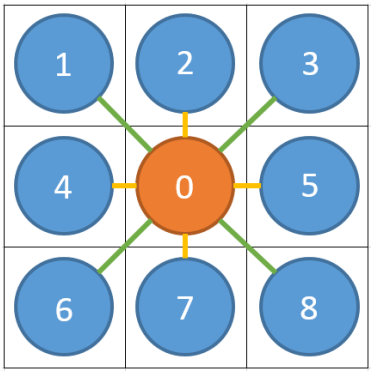


Figure 6.7: Graph representation of an occupancy grid

connections (the yellow lines). In order to simplify the calculation you can store the cost of path as an integer. Calculate the cost by the following equation:

$$c = 5 \cdot \sqrt{(x - x_{par})^2 + (y - y_{par})^2},$$

where c is the calculated cost, x_{par} is the coordinate of 0 th node (see in Fig. 6.7) and x is the coordinate of node on the other side of connection.

After calculating this cost, convert it into integer. Shorter connections has cost 5 and longer connections has 7.

Each node in the graph is stored in the `Node` structure, which contains its position in the grid, information about the occupancy, its parent (the node from which was this node expanded), the cost from start (sum of costs of all edges that were used to get here) and the information, wheter this node is opened, closed or neither.

If the grid would be transfered to the graph this way, the planner would try to find the shortest path and it might go too close to the obstacles, which might be dangerous. The solution is to use higher cost of connections near obstacles (see more in the Section 6.3.4).

6.3.4 Expanding obstacles

Robot is in Dijkstra's algorithm considered as a point so the obstacles must be expanded minimally by the radius of robot. In order to find a safer trajectory through tight spaces, you can expand the obstacles further, but do not block that space, only increase the cost of going through these nodes. You can see an example of expanding obstacles in the Fig. 6.8. The original obstacles are black. The basic expansion (robot is not able to go to these cells) is grey and the cells (nodes in graph) with high cost are orange.

The `cost_modifier` variable in the `Node` structure has predefined values for each case:

-2	Unmapped

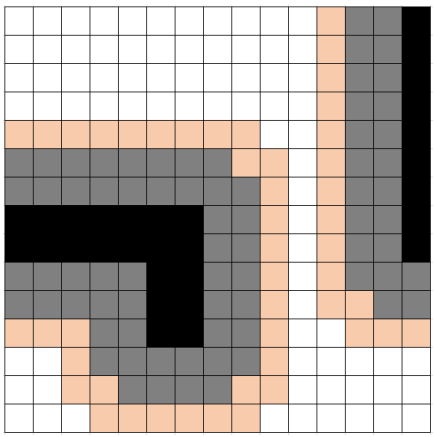


Figure 6.8: Expanded obstacles

-1	Occupied (the robot can no go through this place)
0	Free
$< 1; 10 >$	How much is the cost of going through this node increased ($x \cdot 40\%$, where x is the value in the `occupancy` variable)

As was mentioned above, obstacles are expanded only for the nodes used in planning. It means, that if the planning algorithm reaches a node, which is not yet in the graph (`unordered_map`), it calculates the node's `cost_modifier` and adds it to the graph.

If the value in the occupancy grid is higher than 90, the node is already obstacle so set `cost_modifier` to -1. Otherwise start for cycles that will go through all cells in an occupancy grid in the distance smaller than or equal to `2*robotRadius`. If the obstacle (cell with value higher than 90) is in the distance smaller than or equal to the `robotRadius`, set `cost_modifier` as well to -1. If no obstacle has been found, use the value 0 - free. If the obstacle is in the distance between `robotRadius` and `2*robotRadius`, calculate the cost increase by using the following equation:

$$oc_{(i,j)} = \max(\frac{\sqrt{(i-x)^2 + (j-y)^2} - r}{r}; oc_{(i,j)}),$$

where

x, y	defines the position of an occupied cell in the occupancy grid,
i, j	defines the position of a cell, for which you are calculating the increased cost,
$oc_{(i,j)}$	is an occupancy of cell in (i, j) position and
r	is the radius of robot.

Finally check, if the node is a frontier (unknown space with value between 10 and 90 or equal to -1). If it is a frontier and it is reachable, set the `cost_modifier` to -2.

6.3.5 Expanding nodes

Before you start the Djikstra's algorithm, you need to prepare a method for expanding , i.e. to find neighbour nodes which are possible to reach and calculate their costs. To expand a node look at 8 neigbours (Fig. 6.7). If some of those has the occupancy equal to -1 (occupied), skip them. Nodes with the occupancy equal to -2 (unknown space - you are getting there always from the space you know so it will be a frontier) are not affected by the cost increase from expanding the obstacles. The other nodes have their cost calculated from following equation:

$$c_{(x,y)} = c_{(x_p,y_p)} + k \cdot \frac{100 + oc_{(x,y)} \cdot 40}{100},$$

where

$c_{(x,y)}$ is the calculated cost,

$c_{(x_p,y_p)}$ is the cost of the node from which you are expanding (node 0 in Fig. 6.7),

k is either 5 or 7 (explanation is in the Section 6.3.3) and

$oc_{(x,y)}$ is the value in the variable `occupancy`.

6.3.6 The Dijkstra's algorithm

The open nodes are stored in a priority queue[6]. Before starting the Dijkstra's algorithm, find a node corresponding with the actual position of the robot (according to the transformation). Set up the parent node to a point at itself and cost to 0. Start a cycle which ends if the `occupancy` of an actual node is equal to -2 (the closest frontier is found). Expand the actual node (in the first iteration it is equal to the start node). Then push all nodes gained from the expansion into the priority queue.

Set the `close` variable in the actual node to true (this node may never be expanded again) and pick the first node from the priority queue which is not closed to be the actual node for the next iteration. After the first frontier is found, the cycle ends. Retrace the route to this node by looking at its parent, parent of its parent, etc. until you find the start node.

6.3.7 Cleaning the route

The route from the Djikstra's algorithm contains many superfluous nodes. If r would be the resolution of the map, than the distance between nodes would be in the interval $< r, \sqrt{2} \cdot r >$. A direct line 1 m long would ussualy have ~ 50 nodes (depending on resolution). In order to lower computation requirements, the cleaning algorithm is implemented. It removes all superfluous nodes (the grey ones in Fig. 6.9) from the trajectory.

[6]The Priority queue is a queue, in which are items ordered by their value (using comparator). The one with the lowest value is picked from the queue first.

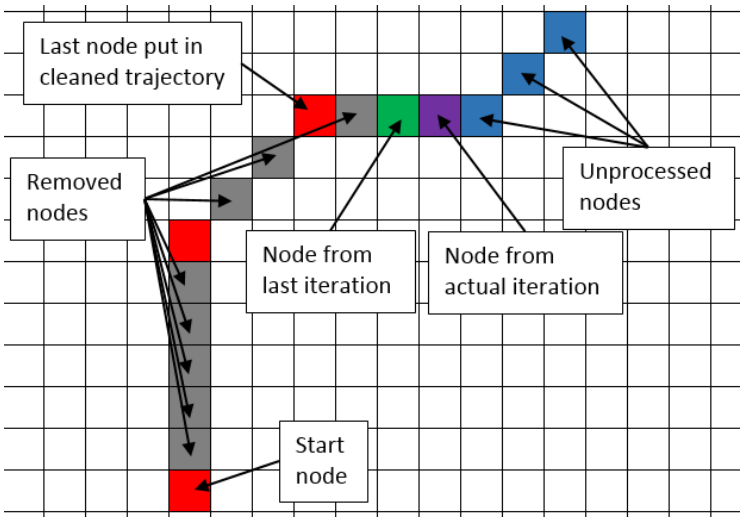


Figure 6.9: Cleaning the trajectory

First push a start node into the trajectory. Keep the knowledge about the last node that was put in the cleaned route and node from the last iteration. Prepare the `angle` variable, which contains angle between the last two nodes (inicialize its value to something that can never be output of `atan2` function - for example 20). Start a cycle, that goes through all nodes in the original trajectory. Skip the first node, which represents the start node. In the beginning of each cycle calculate the angle between the last and actual node by using the following equation:

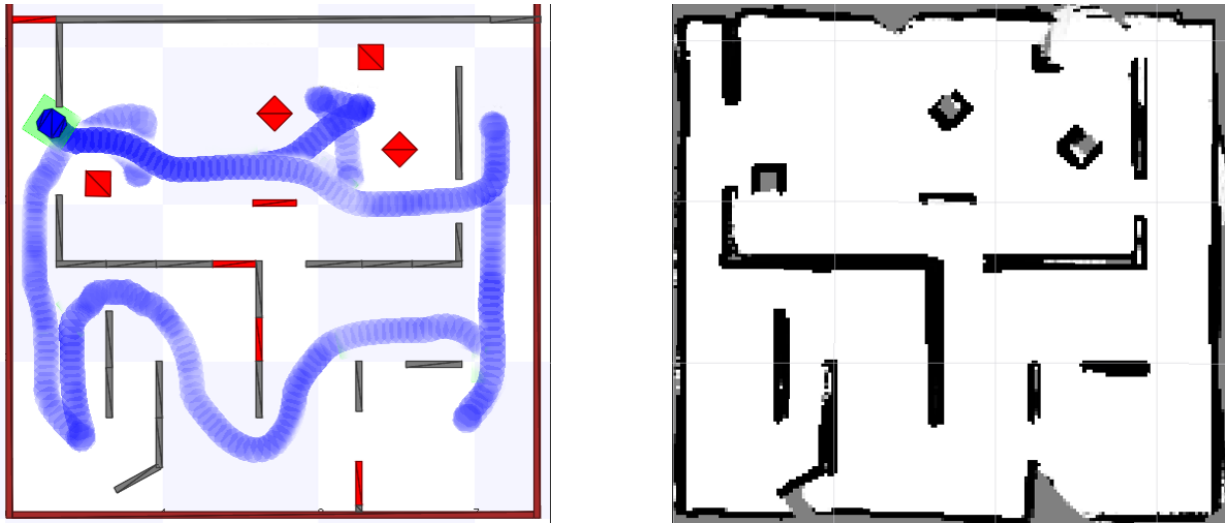
$$\varphi = atan2(y_{actual} - y_{last}, x_{actual} - x_{last})$$

Now check, whether the value in the `angle` variable is 20. If it is true, it means that you are one iteration after the beginning. Save φ into the `angle` variable and continue with the next iteration. If the `angle` is equal or very close ($\sim 10^{-3}$) to φ, continue to the next iteration. An angle between points did not change so the point from the last iteration will be expendable.

If the `angle` is not the same as φ, it means that the direction has changed from the last iteration so push the node from the last iteration into the cleaned route and save φ into the `angle` variable. After the cycle ends, you have finally the clean route to the closest frontier.

6.3.8 Converting the route to the output

Last remaining step is to convert this route into the `Path` message. For the deatiled information about the `Path` messages look at the Section 6.1.1. First you need to fill in the header. Set the `frame_id` to the map frame and the `seq` variable to the number of sent trajectories. Each cell in the data vector has the `position` (x,y,z) and the `orientation` quaternion. Into the `position.x` and `position.y` fill the position of the node. This position can be calculated as x_{robot} and y_{robot} from equation 6.3. The orientation could be set to angle between this and the previous node in the path or to robot's orientation.



(a) The path of the robot (b) The created map

Figure 6.10: Exploration in simulator

In order to use this program with the Navigation Stack[7], add the publishing of goals. The Navigation Stack does not accept the goal in (or near) the unknown space. The parameter `goal_distance` sets up, how far from the frontier (distance on the trajectory) should be the target. The parameter should be bigger then radius of the robot, otherwise the navigation stack would still consider the goal as unreachable.

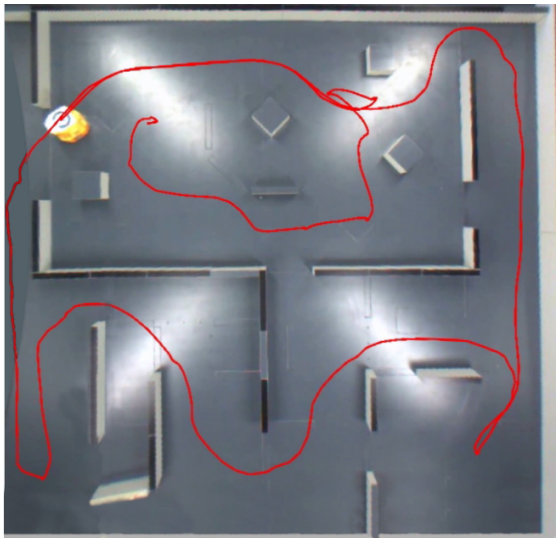
6.4 Exploration results

The whole exploration task (mapping, planning and following) was tested in the simulator first. The results are in the Fig. 6.10.

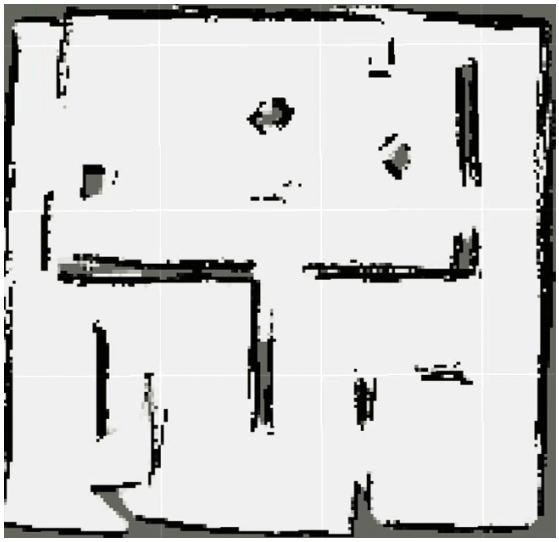
The same test as in the simulator was made in the SyRoTek arena on the real robot. As you can see from the Fig. 6.11, the robot succesfully created the map, but the precision is lower than in the simulator. That might be caused by slower publishing rate of the robot's position and less precise measurements of the rotation by SyRoTek localization system. The record of the whole experiment is published on the SyRoTek website in the Videos section[8].

[7]The Navigation Stack is a ROS package which uses the map and the laser data in order to guide robot safely to a target position.

[8]https://syrotek.felk.cvut.cz/about/videos



(a) The path of the robot



(b) The created map

Figure 6.11: Exploration in SyRoTek

Chapter 7

Exploration with localization

The exploration with localization has the same goal as the previous task, to create a map of the unknown environment, but with the use of the local odometry (measured by the robot itself) instead of the global one and with the use of the Navigation Stack instead of the "Follow the Carrot" algorithm. Simple mapping (like in exploration described in the previous section) can not be used. Local odometry can be after a while very far from the reality. The Navigation Stack can create its own map, but it also does not provide localization. It is necessary to use some external SLAM (simultaneous localization and mapping). There are basicly two choices. The Hector[1] mapping and the Gmapping [23]. The Navigation Stack can also work with the AMCL[2], but it provides only a localization on the static map so it is not useful for an exploration.

The Hector mapping was tested, but the results were much worse than with Gmapping. Whereas the Gmapping localization basically corrects the odometry, the Hector does not use odometry at all and only a map is used for localization. That caused some serious errors in localization so the Gmapping was chosen.

The data from the Gmapping are sent to the planning part of previous exploration, which finds the target frontier (this is done each time the Gmapping publishes refreshed map). The Navigation Stack uses that frontier and the map from the Gmapping to guide the robot safely between the obstacles.

7.1 Navigation Stack

The main goal of the Navigation Stack is to guide a robot safelty to the target position. It works with two trajectories. The first one (global) is planned on the map of the environment and the second one (local) is prepared with the laser scan data (to avoid obstacles, which are not precisely mapped) [24]. The Navigation Stack uses following components to run [25]:

[1] http://wiki.ros.org/hector_mapping
[2] http://wiki.ros.org/amcl

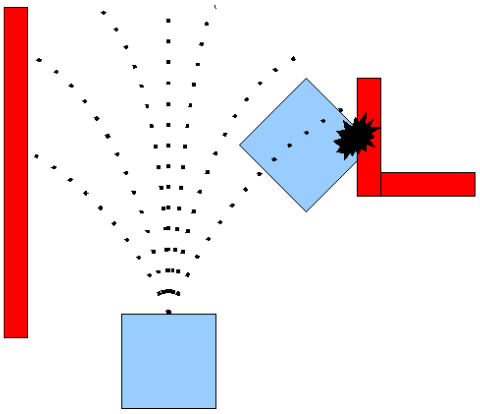


Figure 7.1: Navigation local planner [2]

- A static map of the environment (not necessary, Navigation Stack can create its own map from laser scan data)
- Data from the laser scan (necessary for local planner)
- Position of the robot in the map
- A goal position

The Navigation Stack creates a global plan on the map. This global plan is sent to the local planner. It is transformed from the map frame to the odometry frame. This transformation is necessary, because the map frame does not change continously and that would make robot's movement very shaky. The local planner uses a DWA (Dynamic Window Approach) [26] to create a value function around the robot (represented as an occupancy grid - a local map) from the laser data. The controller uses this value function to calculate linear and angular velocities that are sent to the robot [26]. The illustration of the local planner is in the Fig. 7.1.

7.2 Gmapping

The `gmapping` package in ROS implements a Rao-Blackwellized particle filter[3], that provides simultaneous localization and mapping of the environment [23]. It receives data from the laser scanner, odometry data (in the form of transformation between `/base_link` and `/odom` frames) and transformation between the frame attached to incoming scans and `/base_link`. It creates a map in a form of 2D occupancy grid and provides transformation between `/map` and `/odom` frames. The Gmapping can be used even with a less precise odometry (for example local odometry in SyRoTek - measured by robot itself), because it refines the position by fitting actual laser scan into the occupancy grid.

[3] http://openslam.org/gmapping.html

The Hmapping is very sensitive to transformation timing. It can not be run directly on the SyRoTek server through ssh[4]. Running Gmapping on another machine needs very precise time synchronization with the master[5]. The synchronization needs to be precise, because ROS applications use the time from their machine. The master uses the time from SyRoTek and the Gmapping uses the time from the computer on which it is running.

If the time is not synchronized with the master, the Gmapping discards all transformations (usually because they are too old). Even worse situation might appear, when the time from SyRoTek is ahead of the machine Gmapping running. When this happens, the Gmapping might actually wait for the old transformations and match them with newer laser data. Sometimes even good synchronization is not enough. The Gmapping usually works, but throws away a lot of data (probably due to latency) and the map might be worse or nearly useless.

The way to solve these issues is to resend all the necessary transformations and topics from the machine, on which is running Gmapping. The relay command provided by ROS does not help in this case, because it does not change the time stamp in messages. It is necessary to create an application (in C++), which subscribes to messages, changes the time stamp to the actual time and transmits new messages to the different topic.

In the newer versions of Gmapping the transformation between map and odom frames is published with its time set a bit to the future. This way the transformation is still actual when it arrives. The older versions sends only the actual time. When the messages arrive to the Navigation Stack, it rejects them, because they are too old.

Updating Gmapping to the version from ROS Hydro Medusa solved the problem. That version however has much bigger requirements on processing power, which causes delays in publishing the map. Small improvement can be achieved by using less particles for localization using laser scan. Too low amount of particles would break the localization so a compromise was found (10 particles, 5 iterations and 0.025m accuracy).

7.2.1 Using Navigation Stack with Gmapping

The map for global planner in the Navigation Stack needs to be set to static. It does not mean, that the map can not change. It means, that the map is external and the Navigation Stack should wait, until it is published to the `/map` topic. When the new map is published, it is automatically updated in the global planner. The global frame should be set to `/map` and robot base frame to `/base_link`. The local costmap should not be static. It's global frame should be `/odom`. The path found by the global planner is transfered to this frame by using transformation provided by Gmapping.

The complete setting is in the file `exploration.launch` and included `yaml` files provided on the attached CD.

[4]This option was not available in the time of writing this thesis.

[5]The master is in this case SyRoTek server running the roscore

When the Navigation Stack is connected to the Gmapping, the goals could be obtained from the planning part of the exploration task. That would create autonomous exploration with its own localization. The goal can also be set manually in RVIZ. When the Navigation Stack uses static map of the environment, it consideres all unknown space as obstacles and refuses to go there. This is solved by sending target, which is on the path in specified distance[6] from the frontier.

7.3 Exploration in the SyRoTek

To start an exploration with the local odometry in SyRoTek, modify and resend `LaserScan` and `Twist` messages and all transformations (originally sent from the SyRoTek) from the machine which is running the Gmapping and Navigation Stack to the new topics. This needs to be done because of these reasons:

- The time synchronization mentioned above
- The frame of a laser rangefinder is in SyRoTek placed under the global odometry.
- The minimal and maximal values in messages from the laser scan does not correspond with the size of the data vector. This is caused by cutting values, where the robot would see itself.
- The robot does not follow the angular velocity sent through the twist messages, but rotates approximately at half the speed. The values in the `Twist` messages (commands for the robot) needs to be increased.

The `LaserScan` and `Twist` messages are subscribed by C++ program, which modifies the necessary variables and publishes messages to new topic.

The new transformation tree should look like the right one in the Fig. 7.2. The transformation between `/odom` and `/base_link` should correspond with the transformation between `/arena` and `/syros/base_odom`. The data for publishing new transformation could be obtained from corresponding transformation or from the local odometry messages. The transformation between `/base_link` and `/scan` is static ($[x, y, z] = [0.04, 0, 0], [qx, qy, qz, qw] = \vec{0}$) and the last one between `/map` and `/odom` is published by Gmapping.

7.4 Experiment and results

The Navigation Stack, Gmapping and planning was used in SyRoTek to create a map. The programs used only local odometry. To see the record of this experiment, look at the video

[6] The specified distance should be larger than robot radius. Otherwise it might still be considered as unreachable. Recommended value is 0.15m.

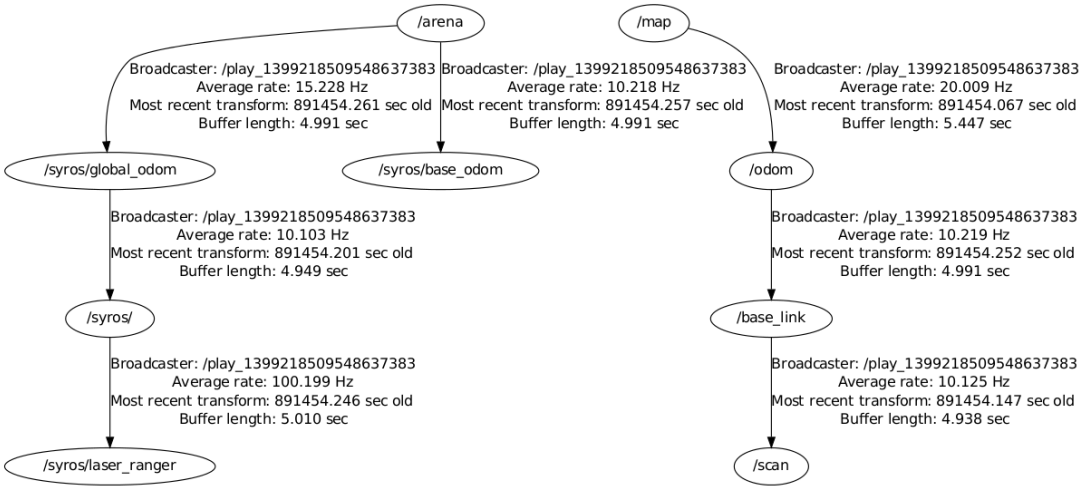


Figure 7.2: Original and new transformation trees

(a) The path in the Arena (b) The created map

Figure 7.3: Exploration with Gmapping and Navigation Stack

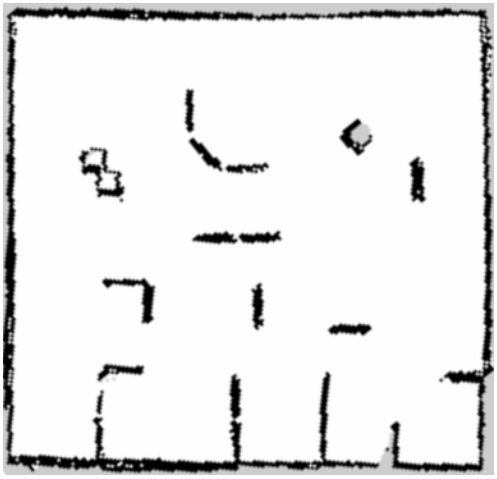


Figure 7.4: The map created by Gmapping with 40 particles and 7 iterations

`Exploration_localization_06_720p.avi`. In the Fig. 7.3 you can see the traveled path and final map.

The movement is not completely smooth. Error in transformations occur sometimes, even though all transformations are resent. These errors causes the Navigation Stack to stop the robot immediately, because it does not know its location. Computational demands of Gmapping caused another stops. The robot needs to wait, until the refreshed map arrives.

Data from the experiment were replayed with slower clock. Gmapping was tested on these data with 40 particles (instead of 10) and 7 iterations (instead of 5). In the Fig. 7.4 is the result. The quality has slightly improved.

Chapter 8

Conclusion

The thesis presents a simplified guide to work with the SyRoTek and the ROS. A reader learns how to create applications, execute them in the SyRoTek as well as in the Stage simulator and visualise the results of experiments in RVIZ.

The serie of demonstration tasks was created so that novice users have the inspiration for their own work. These are the following tasks:

- **Braitenberg vehicle** - The simulation of Braitenberg vehicle was created with the use of data from laser rangefinder. In the experiment the robot was succesfully driving and avoiding obstacles, but in some cases the algorithm was too slow. That lead to implementation of simple obstacle avoidance, which increased angular velocity and decreased linear velocity in dangerous situations.
- **PID controller** - The PID position and orientation controller was implemented. The controller's accuracy was dependent on the accuracy of odometry. Application maintained predefined accuracy (0.01 m and 0.02 rad) in the simulator as simulated odometry is ideally precise. The robot in the SyRoTek was set to move forward by 1 meter, turn around by π radians and repeat this 8 times. In the end, the robot was approximately 10 cm away from the original position.
- **Dead reckoning** - The goal of the dead reckoning was to measure difference between local and global odometry and a distance from the start position. The PID controller was used to controll the robot. This time the robot was going twice through a square trajectory (length of an edge 0.6 m). The difference between local and global odometry was approximately 5 cm in x and y axis and -0.2 radians. The difference from the start position was only 0.4 cm in x axis, but in the y axis it was -4.6 cm and in the rotation 0.1 radians.
- **Wall following** - The wall following algorithm used the laser rangefinder data to drive along the wall in the predefined distance. The robot managed to follow the wall without loosing it in the simulator as well as in the SyRoTek.

- **Trajectory following - pure pursuit** - The pure pursuit algorithm follows the path by calculating a circular trajectory to the control target point (the point on trajectory in a constant distance from the robot). The robot managed to follow the predefined trajectory in the simulator as well as in the SyRoTek, but the real robot was having bigger problems in sharp turns.

- **Autonomous exploration with global odometry** - This task was implemented as three separate programs - mapping, planning and following (using *follow the carrot algorithm*). Communication between these programs was done through ROS topics. This task showed more advanced parts of the ROS system like transformations or parameter server. The result of the experiment was a map of the SyRoTek arena. The robot managed to create a map of the complete arena on its own, but the map was a little inaccurate in some areas due to imperfection of the SyRoTek localization system (see in Fig. 6.11).

- **Autonomous exploration and localization** - In the last task the Gmapping and the Navigation stack were used with the planning part from the previous exploration in order to create an exploration with its own localization. Localization and the map was provided by the Gmapping package which used local odometry (measured by the robot itself). The planning part found frontiers and the Navigation stack was guiding robot safely through the arena. The Gmapping was very demanding on the hardware so it was set to use less particles for the localization. In order to obtain a better map the Gmapping was used on the recorded data from the experiment. These data were replayed with a slower clock.

Commented source codes and video records of the experiments are published on the SyRoTek website[1]. The explanation of tasks in this thesis is only a part of the *SyRoTek Tutorials*[2]. That document is more detailed and contains the explanation of the source code, which would be too long for this thesis.

The Braitenberg vehicle task was also used in the *Practical Guide to the SyRoTek System* [5]. New users of the SyRoTek can now use this thesis as a guide, that will help them to create their own applications.

[1] https://syrotek.felk.cvut.cz/about/codes
[2] https://syrotek.felk.cvut.cz/data/files/

Bibliography

[1] Wikipedia.org user Dcoetzee. File:dijkstra's algorithm.svg, 2007. [Online; accessed 15-February-2014] Available from http://commons.wikimedia.org/wiki/File:Dijkstra%27s_algorithm.svg.

[2] ROS.org. Local plan, 2014. [Online; accessed 4-May-2014] Available from http://wiki.ros.org/base_local_planner?action=AttachFile&do=get&target=local_plan.png Licensed under Creative Commons - Attribution 3.0 http://creativecommons.org/licenses/by/3.0/.

[3] M. Kulich, J. Chudoba, K. Košnar, T. Krajník, J. Faigl, and L. Přeučil. Syrotek - distance teaching of mobile robotics. *Education, IEEE Transactions on*, 56(1):18–23, Feb 2013.

[4] ROS.org. ROS tutorials, 2014. [Online; accessed 18-May-2014] Available from http://wiki.ros.org/ROS/Tutorials.

[5] Intelligent and Mobile Robotics Group. Practical guide to the SyRoTek system. Technical report, CTU FEL, 2014. [Online; accessed 2-May-2014] Available from https://syrotek.felk.cvut.cz/data/files/guide.pdf.

[6] ROS.org. About ROS, 2013. [Online; accessed 1-May-2014] Available from http://www.ros.org/about-ros/.

[7] ROS.org. Introduction, 2014. [Online; accessed 1-May-2014] Available from http://wiki.ros.org/ROS/Introduction.

[8] ROS.org. Understanding ROS nodes, 2014. [Online; accessed 1-May-2014] Available from http://wiki.ros.org/ROS/Tutorials/UnderstandingNodes.

[9] ROS.org. Catkin or rosbuild, 2013. [Online; accessed 1-May-2014] Available from http://wiki.ros.org/catkin_or_rosbuild.

[10] ROS.org. Creating a ROS package, 2013. [Online; accessed 1-May-2014] Available from http://wiki.ros.org/ROS/Tutorials/CreatingPackage.

[11] ROS.org. Writing a simple publisher and subscriber (c++), 2013. [Online; accessed 2-May-2014] Available from http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber(c++).

[12] J. Faigl, J. Chudoba, K. Košnar, M. Kulich, M. Saska, and L. Přeučil. SyRoTek - a robotic system for education. *First International Conference on Robotics in Education, Bratislava*, pages 37–42, 2010.

[13] ROS.org. Rviz user guide, 2014. [Online; accessed 3-May-2014] Available from `http://wiki.ros.org/rviz/UserGuide`.

[14] Wikipedia.org. Braitenberg vehicle, 2012. [Online; accessed 2-May-2014] Available from `http://en.wikipedia.org/wiki/Braitenberg_vehicle`.

[15] Microsoft Developer Network. D3DXQUATERNION structure, 2013. [Online; accessed 6-February-2014] Available from `http://msdn.microsoft.com/en-us/library/bb205402%28v=VS.85%29.aspx`.

[16] J. Faigl, J. Chudoba, M. Kulich, R. Mázl, K. Košnar, L. Přeučil, and P. Štěpán. Syrotek: V003.2 - rámcová definice úloh. Technical report, CTU FEL, 2008.

[17] ROS.org. Parameter server, 2013. [Online; accessed 3-May-2014] Available from `http://wiki.ros.org/Parameter%20Server`.

[18] ROS.org. roscpp/overview/parameter server, 2011. [Online; accessed 3-May-2014] Available from `http://wiki.ros.org/roscpp/Overview/Parameter%20Server`.

[19] ROS.org. roslaunch/xml, 2013. [Online; accessed 3-May-2014] Available from `http://wiki.ros.org/roslaunch/XML`.

[20] ROS.org. TF - ROS wiki, 2014. [Online; accessed 13-April-2014] Available from `http://wiki.ros.org/tf`.

[21] Miroslav Kulich. Environment representation and modeling, 2011. [Online; accessed 13-February-2014] Available from `https://syrotek.felk.cvut.cz/data/files/ECI/mapping.pdf`.

[22] Wikipedia.org. Dijkstra's algorithm, 2014. [Online; accessed 15-February-2014] Available from `http://en.wikipedia.org/wiki/Dijkstra%27s_algorithm`.

[23] ROS.org. Gmapping, 2013. [Online; accessed 13-April-2014] Available from `http://wiki.ros.org/gmapping`.

[24] ROS.org. Navigation, 2013. [Online; accessed 13-April-2014] Available from `http://wiki.ros.org/navigation`.

[25] ROS.org. Navigation tutorials robot setup, 2012. [Online; accessed 13-April-2014] Available from `http://wiki.ros.org/navigation/Tutorials/RobotSetup`.

[26] ROS.org. Base local planner, 2014. [Online; accessed 4-May-2014] Available from `http://wiki.ros.org/base_local_planner`.

Appendix

CD Content

The names of all root directories on CD are listed in table 8.1.

Directory name	Description
/doxygen	Documentation created in Doxygen
/navigation	Launch files for the autonomous exploration task
/src	Source code of applications
/thesis	This thesis in pdf format
/thesissrc	Source code of this thesis
/tutorials	Extended description of the tasks
/video	Video records of the experiments

Table 8.1: CD Content