A

Project Report on

ROS AND GAZEBO BASED MINING ROBOT (Simulation of An Autonomous Robot: SLAM & Navigation)

in
partial fulfilment of the requirements for the degree of

Bachelor of Technology

in

Electronics and Instrumentation Engineering

Submitted by:

Riya Aron (1605232041)

Sudhansh Pathak (1605232050)

Abhishek Kumar (1605251002)

under guidance of

Er. Pradeep kr. Verma

Institute of Engineering and Technology, Lucknow

Dr. A.P.J Abdul Kalam Technical University, Lucknow

CERTIFICATE

This is to certify that Project Report entitled "**ROS AND GAZEBO BASED MINING ROBOT**" which is submitted by Riya Aron, Sudhansh Pathak, Abhishek Kumar in partial fulfilment of the requirement for the award of Bachelor of Technology in Electronics and Instrumentation engineering, in Department of Electronics Engineering of Institute of Engineering and Technology, during the academic year 2019-2020, is a record of the bonafide work carried out by the student during 8^{th} semester under my supervision. The matter embodied in this thesis is original and has not been submitted for the award of any other degree.

Date:

Dr. R.C.S. Chauhan
Project Coordinator

Er. Pradeep Verma
Guide

ACKNOWLEDGEMENT

I wish to express my sincere gratitude to respected Dr. Neelam Srivastava, Head of Department of Electronics Department of Institute of Engineering and Technology, Lucknow for providing me an opportunity to present my Project on "**ROS AND GAZEBO BASED MINING ROBOT**".

My sincere thanks to my project guide Er. Pradeep Verma of Electronics Department, Institute of Engineering and Technology, Lucknow for his valuable guidance and encouragement in carrying out this project.

Last but not the least I wish to avail myself of this opportunity to express a sense of gratitude and love to my friends, sister and my beloved parents for their strength and support.

Date:

RIYA ARON
SUDHANSH PATHAK
ABHISHEK KUMAR

ABSTRACT

The simultaneous localization and map building (SLAM) problem asks if it is possible for an autonomous vehicle to start in an unknown location in an unknown environment and then to incrementally build a map of this environment while simultaneously using this map to compute absolute vehicle location. Starting from the estimation-theoretic foundations, this paper proves that a solution to the SLAM problem is indeed possible. It is then shown that the absolute accuracy of the map and the vehicle location reach a lower bound defined only by the initial vehicle uncertainty. Together, these results show that it is possible for an autonomous vehicle to start in an unknown location in an unknown environment and, using relative observations only, incrementally build a perfect map of the world and to compute simultaneously a bounded estimate of vehicle location. This report also describes a substantial implementation of the SLAM algorithm on a vehicle operating in an outdoor environment to provide relative map observations. This implementation is used to demonstrate how some key issues such as map management and data association can be handled in a practical environment. The results obtained are cross-compared with absolute locations of the map landmarks obtained by surveying. In conclusion, this paper discusses a number of key issues raised by the solution to the SLAM problem including suboptimal map-building algorithms and map management.

CONTENTS

LIST OF FIGURES

Chapter-1

INTRODUCTION

Perception is an important attribute for building any sort of intelligent robot. This aspect is required in a range of robots starting from simple automated vacuum-cleaning robots, self-driving cars to rovers that perform deep space exploration. The perception task can be split primarily into two problems: knowing the environment - area mapping and knowing where the robot is present in the environment - robot localization. To solve the localization problem using any form of on-board sensors a map must be available in the robot from which it can locate its relative position. However, in order to be able to map the surrounding the robot must know its current relative position. This fundamental Simultaneous Localisation and Mapping (SLAM) problem has been at the centre of decades of robotics research.

Since its introduction in the nineties, the Internet has slowly been seeping into everyday life to the point where it is almost taken for granted today. With this increase in connectivity, several core assumptions that formed the pillars of robot-building need to be rethought. When a robot can transmit and receive data at a sufficient rate from a more powerful computing platform, do we really need to have the intelligence to process the data in the robot itself. Several tools have exploited this new paradigm of robot programming. These tools utilize a server and node-based programming approach with multiple independent nodes communicating to each other and the server.

A prominent tool among these being Robot Operating System (ROS), maintained by the Open Source Robotics Foundation. ROS (Robot Operating System) provides libraries and tools to help software developers create robot applications. It provides hardware abstraction, device drivers, libraries, visualizers, message-passing, package management, and more. ROS is licensed under an open source, BSD license.

Through the report, the design decisions and implementations of an attempt to build a low-cost distributed SLAM robotics platform are presented.

1.1 SLAM (Simultaneous Localization and Mapping)

The simultaneous localization and map building (SLAM) problem asks if it is possible for an autonomous vehicle to start in an unknown location in an unknown environment and then to incrementally build a map of this environment while simultaneously using this map to compute absolute vehicle location. The solution to the simultaneous localization and map building (SLAM) problem is, in many respects, a "Holy Grail" of the autonomous vehicle research community. The ability to place an autonomous vehicle at an unknown location in an unknown environment and then have it build a map, using only relative observations of the environment, and then to use this map simultaneously to navigate would indeed make such a robot "autonomous". Thus, the main advantage of SLAM is that it eliminates the need for artificial infrastructures or *a priori* topological knowledge of the environment. A solution to the SLAM problem would be of inestimable value in a range of applications where absolute position or precise map information is unobtainable, including, amongst others, autonomous planetary exploration, subsea autonomous vehicles, autonomous air -borne vehicles, and autonomous all-terrain vehicles in tasks such as mining and construction.

The general SLAM problem has been the subject of substantial research since the inception of a robotics research community and indeed before this in areas such as manned vehicle navigation systems and geophysical surveying. A number of approaches have been proposed to address both the SLAM problem and also more simplified navigation problems where additional map or vehicle location information is made available. Broadly, these approaches adopt one of three main philosophies. The most popular of these is the estimation-theoretic or Kalman filter based approach. The popularity of this approach is due to two main factors. First, it directly provides both a recursive solution to the navigation problem and a means of computing consistent estimates for the uncertainty in vehicle and map landmark locations on the basis of statistical models for vehicle motion and relative landmark observations. Second, a substantial corpus of method and experience has been developed in aerospace, maritime and other navigation applications, from which the autonomous vehicle community can draw. A second philosophy is to eschew the need for absolute position estimates and for precise measures of uncertainty and instead to employ more qualitative knowledge of the relative location of landmarks and vehicles to build maps and guide motion.

The qualitative approach to navigation and the general SLAM problem has many potential advantages over the estimation-theoretic methodology in terms of limiting the need for accurate models and the resulting computational requirements, and in its significant "anthropomorphic appeal". The third, very broad philosophy does away with the rigorous Kalman filter or statistical formalism while retaining an essentially numerical or computational approach to the navigation and SLAM problem. Such approaches include the use of iconic landmark matching, global map registration, bounded regions and other measures to describe uncertainty. Notable are the work by Thrun *et al.* and Yamauchi *et al.*. Thrun *et al.* use a bayesian approach to map building that does not assume Gaussian probability distributions as required by the Kalman filter. This technique, while very effective for localization with respect to maps, does not lend itself to provide an incremental solution to SLAM where a map is gradually built as information is received from sensors. Yamauchi *et al,* use a evidence grid approach that requires that the environment is decomposed to a number of cells.

An estimation-theoretic or Kalman filter based approach to the SLAM problem is adopted in this project. A major advantage of this approach is that it is possible to develop a complete proof of the various properties of the SLAM problem and to study systematically the evolution of the map and the uncertainty in the map and vehicle location. A proof of existence and

convergence for a solution of the SLAM problem within a formal estimation-theoretic framework also encompasses the widest possible range of navigation problems and implies that solutions to the problem using other approaches are possible.

The study of estimation-theoretic solutions to the SLAM problem within the robotics community has an interesting history. Initial work by Smith *et al.*and Durrant-Whyte established a statistical basis for describing relationships between landmarks and manipulating geometric uncertainty. A key element of this work was to show that there must be a high degree of correlation between estimates of the location of different landmarks in a map and that indeed these correlations would grow to unity following successive observations. At the same time Ayache and Faugeras and Chatila and Laumond were undertaking early work in visual navigation of mobile robots using Kalman filter-type algorithms. These two strands of research had much in common and resulted soon after in the key paper by Smith, Self and Cheeseman. This project shows that as a mobile robot moves through an unknown environment taking relative observations of landmarks, the estimates of these landmarks are all necessarily correlated with each other because of the common error in estimated vehicle location. This theory was followed by a series of related work developing a number of aspects of the essential SLAM problem. The main conclusion of this work was twofold. First, accounting for correlations between landmarks in a map is important if filter consistency is to be maintained. Second, that a full SLAM solution requires that a state vector consisting of all states in the vehicle model and all states of every landmark in the map needs to be maintained and updated following each observation if a complete solution to the SLAM problem is required. The consequence of this in any real application is that the Kalman filter needs to employ a huge state vector (of order the number of landmarks maintained in the map) and is in general, computationally intractable. Crucially, this work did not look at the convergence properties of the map or its steady-state behaviour. Indeed, it was widely assumed at the time that the estimated map errors would not converge and would instead execute a random walk behaviour with unbounded error growth. Given the computational complexity of the SLAM problem and without knowledge of the convergence behaviour of the map, a series of approximations to the full SLAM solution were proposed which assumed that the correlations between landmarks could be minimized or eliminated thus reducing the full filter to a series of decoupled landmark to vehicle filters, Leonard and Durrant-Whyte. Also, for these reasons theoretical work on the full estimation-theoretic SLAM problem largely ceased, with effort instead being expended in map-based navigation and alternative theoretical approaches to the SLAM problem.

It assumes an autonomous vehicle (mobile robot) equipped with a sensor capable of making measurements of the location of landmarks relative to the vehicle. The landmarks may be artificial or natural and it is assumed that the signal processing algorithms are available to detect these landmarks. The vehicle starts at an unknown location with no knowledge of the location of landmarks in the environment. As the vehicle moves through the environment (in a stochastic manner) it makes relative observations of the location of individual landmarks.

The above theory proves the following three results.

1. The determinant of any submatrix of the map covariance matrix decreases monotonically as observations are successively made.
2. In the limit as the number of observations increases, the landmark estimates become fully correlated.
3. In the limit, the covariance associated with any single landmark location estimate is determined only by the initial covariance in the vehicle location estimate.

These three results describe, in full, the convergence properties of the map and its steady state behaviour. In particular they show the following.

1. The entire structure of the SLAM problem critically depends on maintaining complete knowledge of the cross correlation between landmark estimates. Minimizing or ignoring cross correlations is precisely contrary to the structure of the problem.
2. As the vehicle progresses through the environment the errors in the estimates of any pair of landmarks become more and more correlated, and indeed never become less correlated.
3. In the limit, the errors in the estimates of any pair of landmarks become fully correlated. This means that given the exact location of any one landmark, the location of any other landmark in the map can also be determined with absolute certainty.
4. As the vehicle moves through the environment taking observations of individual landmarks, the error in the estimates of the relative location between different landmarks reduces monotonically to the point where the map of relative locations is known with absolute precision.
5. As the map converges in the above manner, the error in the absolute location of every landmark (and thus the whole map) reaches a lower bound determined only by the error that existed when the first observation was made.

Thus, a solution to the general SLAM problem exists and it is indeed possible to construct a perfectly accurate map and simultaneously compute vehicle position estimates without any prior knowledge of vehicle or landmark locations.

This theory makes three principal contributions to the solution of the SLAM problem. First, it proves three key convergence properties of the full SLAM filter. Second, it elucidates the true structure of the SLAM problem and shows how this can be used in developing consistent SLAM algorithms. Finally, it demonstrates and evaluates the implementation of the full SLAM algorithm in an outdoor environment using a millimetre wave (MMW) radar sensor.

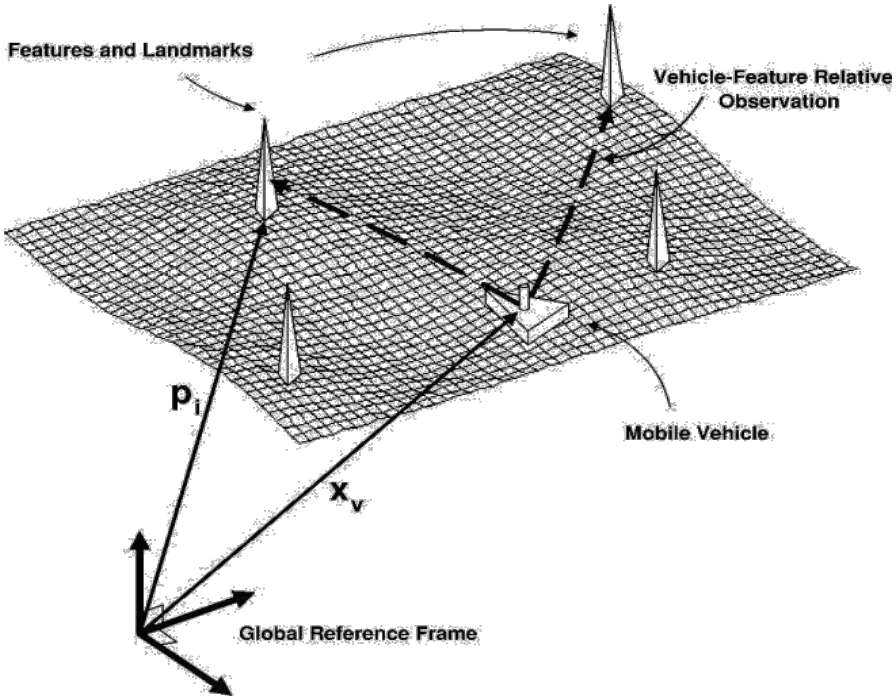


Fig1.1: A vehicle taking relative measurements to environment landmarks

Chapter-2

SOFTWARE DESCRIPTION: ROS

2.1 About ROS

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 behaviour across a wide variety of robotic platforms.

Why? Because creating a truly robust, general-purpose robot software is hard. From the robot's perspective, problems that seem trivial to humans often vary wildly between instances of tasks and environments. Dealing with these variations is so hard that no single individual, laboratory, or institution can hope to do it on their own.

Fig. 2.1: ROS

As a result, ROS was built from the ground up to encourage collaborative robotics software development. For example, one laboratory might have experts in mapping indoor environments and could contribute a world-class system for producing maps. Another group might have experts at using maps to navigate, and yet another group might have discovered a computer vision approach that works well for recognizing small objects in clutter. ROS was designed specifically for groups like these to collaborate and build upon each other's work, as is described throughout this site.

Fig. 2.2: ROS Description

2.1.1 History

ROS is a large project with many ancestors and contributors. The need for an open-ended collaboration framework was felt by many people in the robotics research community, and many projects have been created towards this goal.

Various efforts at Stanford University in the mid-2000s involving integrative, embodied AI, such as the Stanford AI Robot (STAIR) and the Personal Robots (PR) program, created in-house prototypes of flexible, dynamic software systems intended for robotics use. In 2007, Willow Garage, a nearby visionary robotics incubator, provided significant resources to extend these concepts much further and create well-tested implementations. The effort was boosted by countless researchers who contributed their time and expertise to both the core ROS ideas and to its fundamental software packages. Throughout, the software was developed in the open using the permissive BSD open-source license and gradually has become a widely-used platform in the robotics research community.

From the start, ROS was developed at multiple institutions and for multiple robots, including many institutions that received PR2 robots from Willow Garage. Although it would have been far simpler for all contributors to place their code on the same servers, over the years, the "federated" model has emerged as one of the great strengths of the ROS ecosystem. Any group can start their own ROS code repository on their own servers, and they maintain full ownership and control of it. They don't need anyone's permission. If they choose to make their repository publicly available, they can receive the recognition and credit they deserve for their achievements, and benefit from specific technical feedback and improvements like all open-source software projects.

The ROS ecosystem now consists of tens of thousands of users worldwide, working in domains ranging from tabletop hobby projects to large industrial automation systems.

2.1.2 Core Components

While we cannot provide an exhaustive list of what is in the ROS ecosystem, we can identify some of the core parts of ROS and talk about their functionality, technical specifications, and quality in order to give a better idea of what ROS can contribute to this project.

2.1.2.1 Communications Infrastructure

At the lowest level, ROS offers a message passing interface that provides inter-process communication and is commonly referred to as a middleware.

The ROS middleware provides these facilities:

- publish/subscribe anonymous message passing
- recording and playback of messages
- request/response remote procedure calls
- distributed parameter system

- **Message Passing**

  A communication system is often one of the first needs to arise when implementing a new robot application. ROS's built-in and well-tested messaging system saves our time by managing the details of communication between distributed nodes via the anonymous publish/subscribe mechanism. Another benefit of using a message passing system is that it forces us to implement clear interfaces between the nodes in our system, thereby improving encapsulation and promoting code reuse. The structure of these message interfaces is defined in the message IDL (Interface Description Language).

- **Recording and Playback of Messages**

  Because the publish/subscribe system is anonymous and asynchronous, the data can be easily captured and replayed without any changes to code. Say you have Task A that reads data from a sensor, and you are developing Task B that processes the data produced by Task A. ROS makes it easy to capture the data published by Task A to a file, and then republish that data from the file at a later time. The message-passing abstraction allows Task B to be agnostic with respect to the source of the data, which

could be Task A or the log file. This is a powerful design pattern that can significantly reduce our development effort and promote flexibility and modularity in our system.

- **Remote Procedure Calls**

  The asynchronous nature of publish/subscribe messaging works for many communication needs in robotics, but sometimes you want synchronous request/response interactions between processes. The ROS middleware provides this capability using services. Like topics, the data being sent between processes in a service call are defined with the same simple message IDL.

- **Distributed Parameter System**

  The ROS middleware also provides a way for tasks to share configuration information through a global key-value store. This system allows you to easily modify your task settings, and even allows tasks to change the configuration of other tasks.

2.1.2.2 Robot-Specific Features

In addition to the core middleware components, ROS provides common robot-specific libraries and tools that will get our robot up and running quickly. Here are just a few of the robot-specific capabilities that ROS provides:

- Standard Message Definitions for Robots
- Robot Geometry Library
- Robot Description Language
- Preemptable Remote Procedure Calls
- Diagnostics
- Pose Estimation
- Localization
- Mapping
- Navigation

- **Standard Robot Messages**

  Years of community discussion and development have led to a set of standard message formats that cover most of the common use cases in robotics. There are message definitions for geometric concepts like poses, transforms, and vectors; for sensors like cameras, IMUs and lasers; and for navigation data like odometry, paths, and maps; among many others. By using these standard messages in our application, our code will interoperate seamlessly with the rest of the ROS ecosystem, from development tools to libraries of capabilities.

- ## Robot Geometry Library

A common challenge in many robotics projects is keeping track of where different parts of the robot are with respect to each other. For example, if you want to combine data from a camera with data from a laser, you need to know where each sensor is, in some common frame of reference. This issue is especially important for humanoid robots with many moving parts. We address this problem in ROS with the tf (transform) library, which will keep track of where everything is in your robot system.

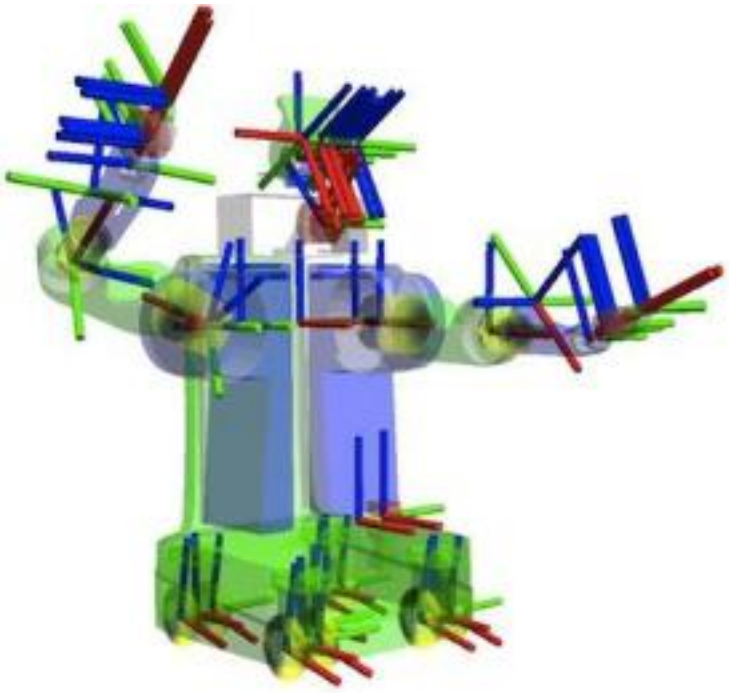


Fig. 2.3: Humanoid Robot Model

Designed with efficiency in mind, the tf library has been used to manage coordinate transform data for robots with more than one hundred degrees of freedom and update rates of hundreds of Hertz. The tf library allows us to define both static transforms, such as a camera that is fixed to a mobile base, and dynamic transforms, such as a joint in a robot arm. We can transform sensor data between any pair of coordinate frames in the system. The tf library handles the fact that the producers and consumers of this information may be distributed across the network, and the fact that the information is updated at varying rates.

- ## Robot Description Language

Another common robotics problem that ROS solves for us is how to describe our robot in a machine-readable way. ROS provides a set of tools for describing and modelling our robot so that it can be understood by the rest of our ROS system, including tf, robot_state_publisher, and rviz. The format for describing your robot in ROS is URDF (Unified Robot Description Format), which consists of an XML document in which we describe the physical properties of robot, from the lengths of limbs and sizes of wheels to the locations of sensors and the visual appearance of each part of the robot.

Once defined in this way, our robot can be easily used with the tf library, rendered in three dimensions for nice visualizations, and used with simulators and motion planners.

- **Preemptable Remote Procedure Calls**

  While topics (anonymous publish/subscribe) and services (remote procedure calls) cover most of the communication use cases in robotics, sometimes we need to initiate a goal-seeking behaviour, monitor its progress, be able to pre-empt it along the way, and receive notification when it is complete. ROS provides actions for this purpose. Actions are like services except they can report progress before returning the final response, and they can be pre-empted by the caller. So, for example, we can instruct our robot to navigate to some location, monitor its progress as it attempts to get there, stop or redirect it along the way, and be told when it has succeeded (or failed). An action is a powerful concept that is used throughout the ROS ecosystem.

- **Diagnostics**

  ROS provides a standard way to produce, collect, and aggregate diagnostics about our robot so that, at a glance, we can quickly see the state of our robot and determine how to address issues as they arise.

Fig. 2.4: Aggregate Diagnostics

- **Pose Estimation, Localization, and Navigation**

  ROS also provides some "batteries included" capabilities that help us get started on our robotics project. There are ROS packages that solve basic robotics problems like pose estimation, localization in a map, building a map, and even mobile navigation.

  Whether you are an engineer looking to do some rapid research and development, a robotics researcher wanting to get your research done in a timely fashion, or a hobbyist looking to learn more about robotics, these out-of-the-box capabilities will help you do more, with less effort.

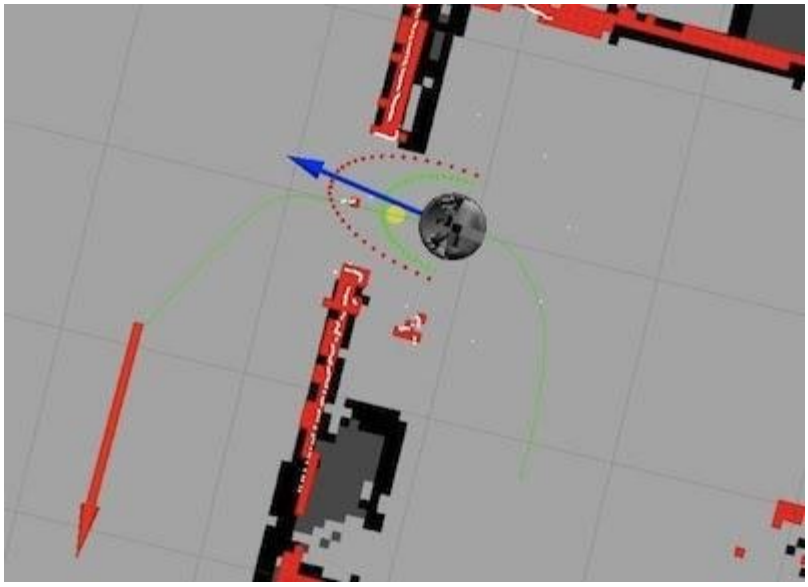


Fig. 2.5: Pose Estimation & Localization

2.1.3 Tools

One of the strongest features of ROS is the powerful development toolset. These tools support introspecting, debugging, plotting, and visualizing the state of the system being developed. The underlying publish/subscribe mechanism allows you to spontaneously introspect the data flowing through the system, making it easy to comprehend and debug issues as they occur. The ROS tools take advantage of this introspection capability through an extensive collection of graphical and command line utilities that simplify development and debugging.

2.1.3.1 Command-Line Tools

Do you spend all of your time remotely logged into a robot? ROS can be used 100% without a GUI. All core functionality and introspection tools are accessible via one of our more than 45 command line tools. There are commands for launching groups of nodes; introspecting topics, services, and actions; recording and playing back data; and a host of other situations. If you prefer to use graphical tools, rviz and rqt provide similar (and extended) functionality.

2.1.3.2 rviz

Perhaps the most well-known tool in ROS, rviz provides general purpose, three-dimensional visualization of many sensor data types and any URDF-described robot. rviz can visualize many of the common message types provided in ROS, such as laser scans, three-dimensional point clouds, and camera images. It also uses information from the tf library to show all of the sensor data in a common coordinate frame of your choice, together with a three-dimensional rendering of your robot. Visualizing all of your data in the same application not only looks impressive, but also allows you to quickly see what your robot sees, and identify problems such as sensor misalignments or robot model inaccuracies.

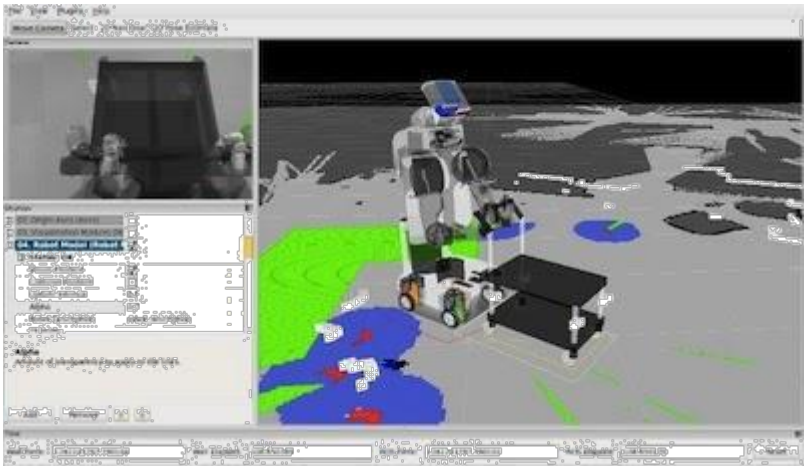


Fig. 2.6: rviz Visualization

2.1.3.3 rqt

ROS provides rqt, a Qt-based framework for developing graphical interfaces for your robot. You can create custom interfaces by composing and configuring the extensive library of built-in rqt plugins into tabbed, split-screen, and other layouts. You can also introduce new interface components by writing your own rqt plugins .

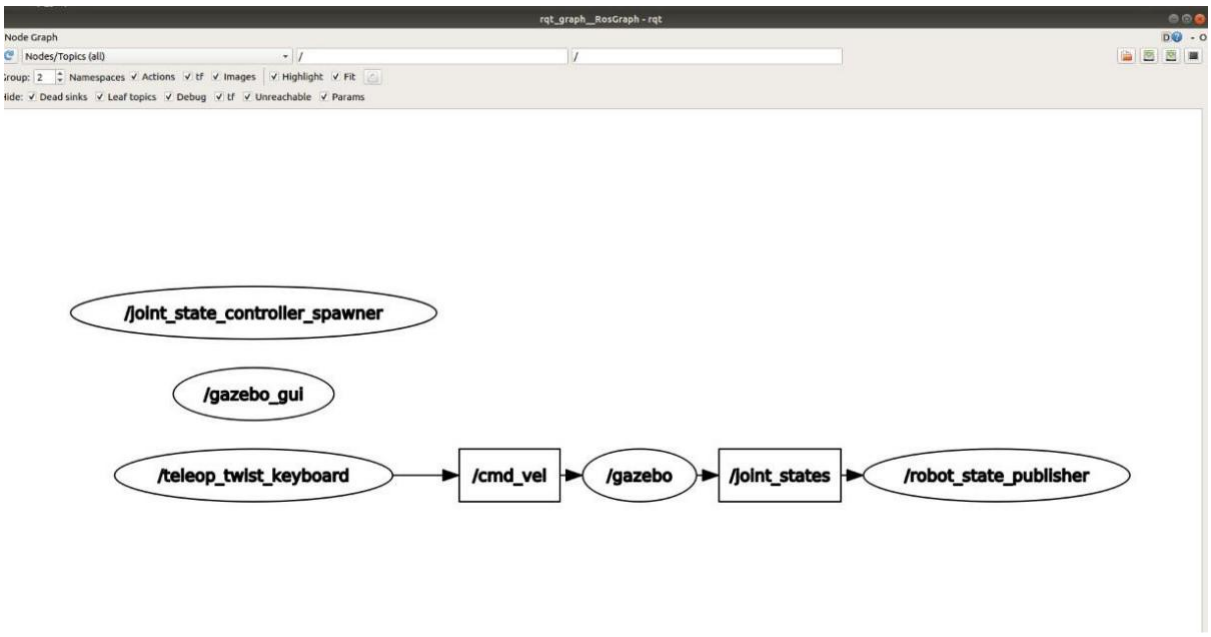


Fig. 2.7: rqt_graph

The rqt_graph plugin provides introspection and visualization of a live ROS system, showing nodes and the connections between them, and allowing you to easily debug and understand your running system and how it is structured.

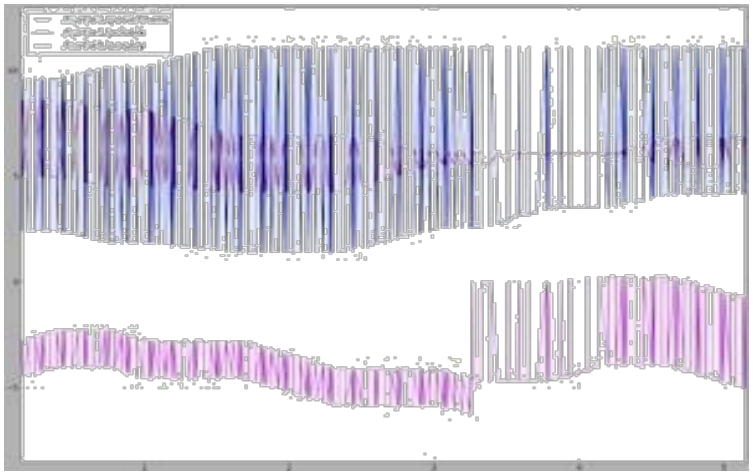


Fig. 2.8: rqt Plot Plugin

With the rqt_plot plugin, you can monitor encoders, voltages, or anything that can be represented as a number that varies over time. The rqt_plot plugin allows you to choose the plotting backend (e.g., matplotlib, Qwt, pyqtgraph) that best fits your needs.

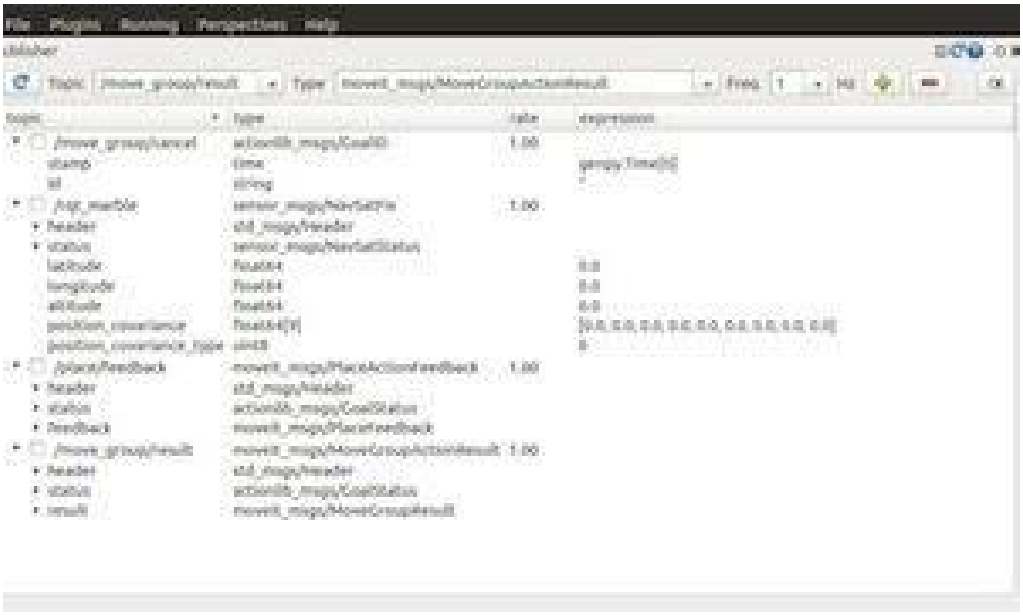


Fig. 2.9:

For monitoring and using topics, you have the rqt_topic andrqt_publisher plugins. The former lets you monitor and introspect any number of topics being published within the system. The latter allows you to publish your own messages to any topic, facilitating ad hoc experimentation with your system. For data logging and playback, ROS uses the bag format. Bag files can be created and accessed graphically via the rqt_bag plugin. This plugin can record data to bags, play back selected topics from a bag, and visualize the contents of a bag, including display of images and plotting of numerical values over time.

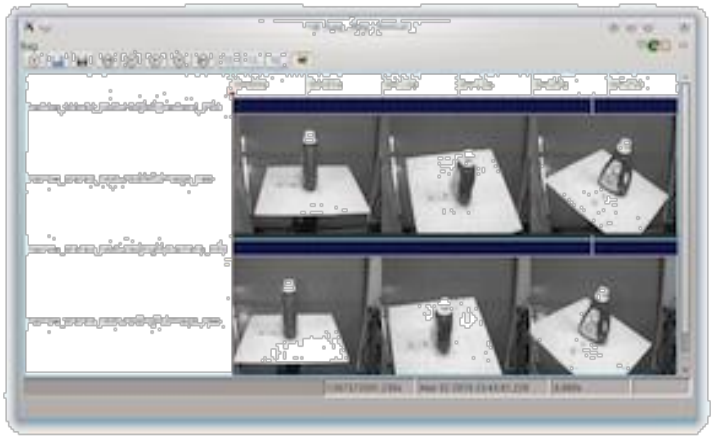


Fig. 2.10:

2.1.4 Integration with Other Libraries

Do you want to combine OpenCV and MoveIt to make your robot detect and pick up an object? ROS has covered, providing seamless integration with these and other popular open source projects, along with a message passing system that allows you to easily swap out different data sources, from live sensors to log files.

2.1.4.1 Gazebo

Gazebo is a 3D indoor and outdoor multi-robot simulator, complete with dynamic and kinematic physics, and a pluggable physics engine. Integration between ROS and Gazebo is provided by a set of Gazebo plugins that support many existing robots and sensors. Because the plugins present the same message interface as the rest of the ROS ecosystem, you can write ROS nodes that are compatible with simulation, logged data, and hardware. You can develop your application in simulation and then deploy to the physical robot with little or no changes in your code.

Fig. 2.11 Gazebo

Gazebo supports the following:

- Designing of robot models
- Rapid prototyping and testing of algorithms
- Regression testing using realistic scenarios
- Simulation of indoor and outdoor environments
- Simulation of sensor data for laser range finders, 2D/3D cameras, kinect-style sensors, contact sensors, force-torque
- Advanced 3D objects and environments utilizing Object-Oriented Graphics Rendering Engine (OGRE)
- Several high-performance physics engines (Open Dynamics Engine (ODE), Bullet, Simbody, and Dynamic Animation and Robotics Toolkit (DART)) to model the real-world dynamics.

Supported Vehicles: Quad (Iris and Solo, Hex (Typhoon H480), Generic quad delta VTOL, Tail sitter, Plane, Rover, Submarine.

Quadrotor with Optical flow

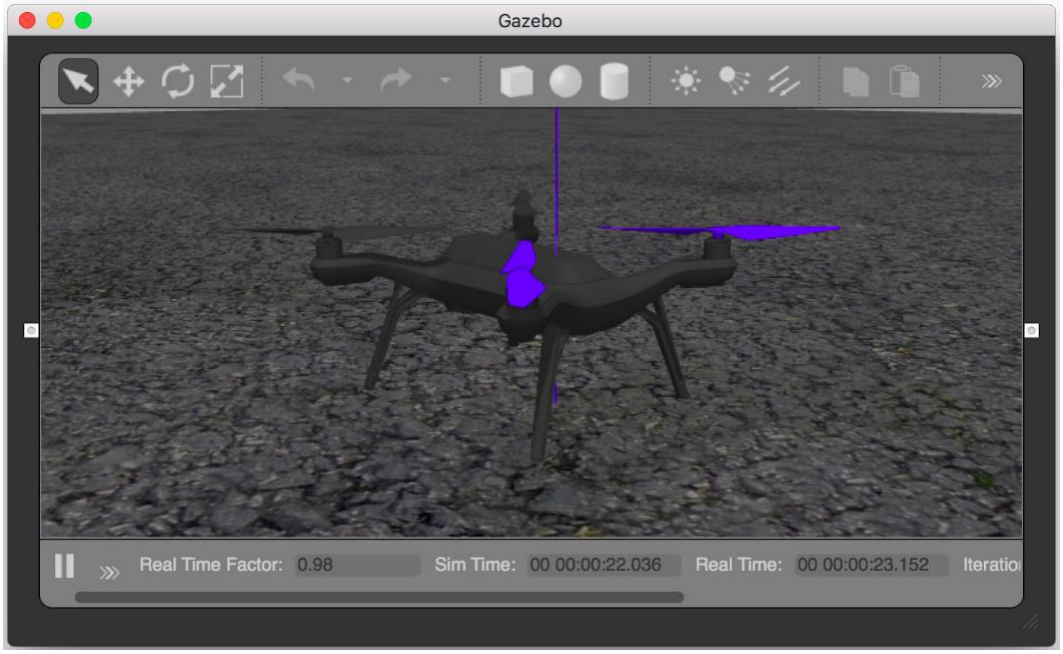


Fig. 2.12 Quadrotor with optical flow

2.1.4.2 OpenCV

OpenCV is the premier computer vision library, used in academia and in products around the world. OpenCV provides many common computer vision algorithms and utilities that you can use and build upon. ROS provides tight integration with OpenCV, allowing users to easily feed data published by cameras of various types into OpenCV algorithms, such as segmentation and tracking. ROS builds on OpenCV to provide libraries such as image_pipeline, which can be used for camera calibration, monocular image processing, stereo image processing, and depth image processing. If your robot has cameras connected through USB, Firewire, or Ethernet, ROS and OpenCV will make your life easier.

Fig. 2.13: OpenCV

2.1.4.3 Point Cloud Library

PCL, the Point Cloud Library, is a perception library focused on the manipulation and processing of three-dimensional data and depth images. PCL provides many point cloud algorithms, including filtering, feature detection, registration, kd-trees, octrees, sample consensus, and more. If you are working with a three-dimensional sensor like the Microsoft

Fig. 2.14: PCL

Kinect or a scanning laser, then PCL and ROS will help you collect, transform, process, visualize, and act upon that rich 3D data.

2.1.4.4 MoveIT

MoveIt is a motion planning library that offers efficient, well-tested implementations of state of the art planning algorithms that have been used on a wide variety of robots, from simple wheeled platforms to walking humanoids.

Whether you want to apply an existing algorithm or develop your own approach, MoveIt has what you need for motion planning. Through its integration with ROS, MoveIt can be used with any ROS-supported robot. Planning data can be visualized with rviz and rqt plugins, and plans can be executed via the ROS control system.

Fig. 2.15: MoveIT

2.2 ROS Industrial

ROS-Industrial is an open-source project that extends the advanced capabilities of ROS to manufacturing automation and robotics. The ROS-Industrial repository includes interfaces for common industrial manipulators, grippers, sensors, and device networks. It also provides software libraries for automatic 2D/3D sensor calibration, process path/motion planning, applications like Scan-N-Plan, developer tools like the Qt Creator ROS Plugin, and training curriculum that is specific to the needs of manufacturers. ROS-I is supported by an international Consortium of industry and research members.

Fig. 2.16: ROS Industrial

2.2.1 ROS Installation Options

There is more than one ROS distribution supported at a time. Some are older releases with long term support, making them more stable, while others are newer with shorter support life times, but with binaries for more recent platforms and more recent versions of the ROS packages that make them up.

2.2.1.1 What is a Distribution?

A ROS distribution is a versioned set of ROS packages. These are akin to Linux distributions (e.g. Ubuntu). The purpose of the ROS distributions is to let developers work against a relatively stable codebase until they are ready to roll everything forward. Therefore, once a distribution is released, we try to limit changes to bug fixes and non-breaking improvements for the core packages (everything under ros-desktop-full). And generally, that applies to the whole community, but for "higher" level packages, the rules are less strict, and so it falls to the maintainers of a given package to avoid breaking changes.

We capture the components that make up a distribution in our rosdistro format and it allows for multiple distributions. There are many different types of robots with different needs, and we anticipate that parts of the community may put together their own distributions in the future to better target these platforms.

2.2.1.2 Which distribution to use

Use Melodic for Gazebo 9, Indigo or later for using OpenCV3. For Linux system with Ubuntu 16.04 Kinetic Kame is required.

2.2.1.3 Ubuntu install of ROS Melodic

We are building Debian packages for several Ubuntu platforms, listed below. These packages are more efficient than source-based builds and are our preferred installation method for Ubuntu. Note that there are also packages available from Ubuntu upstream. Please see Upstream Packages to understand the difference.

- **Installation**
  - ## Configure your Ubuntu repositories

    Configure your Ubuntu repositories to allow "restricted," "universe," and "multiverse." You can follow the Ubuntu guide for instructions on doing this.
  - ## Setup your sources.list

    Setup your computer to accept software from packages.ros.org.

    sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(lsb_release -sc) main" > /etc/apt/sources.list.d/ros-latest.list'
  - ## Set up your keys

    sudo apt-key adv --keyserver 'hkp://keyserver.ubuntu.com:80' --recv-keyC1CF6E31E6BADE8868B172B4F42ED6FBAB17C654

    If you experience issues connecting to the key server, you can try substituting hkp://pgp.mit.edu:80 or hkp://keyserver.ubuntu.com:80 in the previous command.
  - ## Installation

    First, make sure your Debian package index is up-to-date:

    sudo apt update

    There are many different libraries and tools in ROS. We provided four default configurations to get you started. You can also install ROS packages individually.

    In case of problems with the next step, you can use following repositories instead of the ones mentioned above ros-shadow-fixed.
  - ## Desktop-Full Install: (Recommended)

    ROS, rqt, rviz, robot-generic libraries, 2D/3D simulators and 2D/3D

    perception *sudo apt install ros-melodic-desktop-full* To find available

    packages, use:

    apt search ros-melodic
  - ## Environment setup

    It's convenient if the ROS environment variables are automatically added to your bash session every time a new shell is launched:

    echo "source /opt/ros/melodic/setup.bash" >> ~/.bashrc source ~/.bashrc

    If you have more than one ROS distribution installed, ~/.bashrc must only source the setup.bash for the version you are currently using.
    If you just want to change the environment of your current shell, instead of the above you can type:

source /opt/ros/melodic/setup.bash

- ## Dependencies for building packages

  Up to now you have installed what you need to run the core ROS packages. To create and manage your own ROS workspaces, there are various tools and requirements that are distributed separately. For example, rosinstall is a frequently used command-line tool that enables you to easily download many source trees for ROS packages with one command.

  To install this tool and other dependencies for building ROS packages, run:

  sudo apt install python-rosdep python-rosinstall python-rosinstall-generator python-wstool build-essential

- ## Initialize rosdep

  Before you can use many ROS tools, you will need to initialize rosdep. rosdep enables you to easily install system dependencies for source you want to compile and is required to run some core components in ROS. If you have not yet installed rosdep, do so as follows:

  sudo apt install python-rosdep

  With the following, you can initialize rosdep.

  sudo rosdep init
  rosdep update

- ## Managing Your Environment

  During the installation of ROS, you will see that you are prompted to source one of several setup.*sh files, or even add this 'sourcing' to your shell startup script. This is required because ROS relies on the notion of combining spaces using the shell environment. This makes developing against different versions of ROS or against different sets of packages easier.

  If you are ever having problems finding or using your ROS packages make sure that you have your environment properly setup. A good way to check is to ensure that environment variables like ROS_ROOT and ROS_PACKAGE_PATH are set:

  printenv | grep ROS

  If they are not then you might need to 'source' some setup.*sh files.

  Note: Throughout the report, we will see references to rosbuild and catkin. These are the two available methods for organizing and building your ROS code. rosbuild is not recommended or maintained anymore but kept for legacy. catkin is the recommended way to organise your code, it uses more standard CMake conventions and provides more flexibility especially for people wanting to integrate external code bases or who want to release their software.

  If you just installed ROS from apt on Ubuntu then you will have setup.*sh files in '/opt/ros/<distro>/', and you could source them like so:

source /opt/ros/<distro>/setup.bash

Using the short name of your ROS distribution instead of <distro>
If you installed ROS Kinetic, that would be:

source /opt/ros/kinetic/setup.bash

You will need to run this command on every new shell you open to have access to the ROS commands, unless you add this line to your .bashrc. This process allows you to install several ROS distributions (e.g. indigo and kinetic) on the same computer and switch between them.

On other platforms you will find these *setup.*sh* files wherever you installed ROS.

- **System Structure**

ROS is a software suite which allows for quick and easy building of autonomous robotic systems. ROS should be considered as a set of tools for creating new solutions or adjusting already existing ones. A major advantage of this system is a great set of drivers and implemented algorithms widely used in robotics.

- **Nodes**

Base unit in ROS is called a node. Nodes are in charge of handling devices or computing algorithms- each node for separate task. Nodes can communicate with each other using topics or services. ROS software is distributed in packages. Single package is usually developed for performing one type of task and can contain one or multiple nodes.

- **Topics**

In ROS, topic is a data stream used to exchange information between nodes. They are used to send frequent messages of one type. This could be a sensor readout or motor goal speed. Each topic is registered under the unique name and with defined message type. Nodes can connect with it to publish messages or subscribe to them. For a given topic, one node cannot publish and subscribe to it at the same time, but there is no restrictions in the number of different nodes publishing or subscribing.

- **Services**

Communication by services resemble client-server model. In this mode one node (the server) registers service in the system. Later any other node can ask that service and get response. In contrast to topics, services allow for two-way communication, as a request can also contain some data.

- **Basic tools**

While working with ROS there are some tools that are most useful. They are intended to examine nodes and topics.

- **rosnode**

Rosnode is a command line application for examining which nodes are registered in the system and also checking their statuses.

Using the application looks as follows:

rosnode command [node_name]

Command could be:

- list - display list of running nodes
- info - display info regarding selected node
- kill - stop selected node

Detailed info could be found in ROS documentation.

- **rostopic**

  Rostopic is a command line application for examining which topics are already being published and subscribed, checking details of the selected topic or reading messages being sent in it.

  Using the application looks as follows:

  rostopic command [topic_name]

  Command could be:

  - list - display list of topics
  - info - display info regarding selected topic
  - echo - display messages published in the topic

  Detailed info could be found in ROS documentation.

- **rqt_graph**

  rqt_graph is a graphical tool for visualization of data flow across different nodes in the system.

  Using the application looks as follows:

  rqt_graph

Chapter-3

STATE ESTIMATION AND LOCALIZATION

Localization is the method by which we determine the position and orientation of a vehicle within the world. If we want to drive autonomously, we certainly need to know where we are. To accomplish this, we can use state estimation. This is the process of computing a physical quantity like position from a set of measurements.

Since any real-world measurements will be imprecise, we will develop methods that try to find the best or optimal value given some assumptions about our sensors and the external world. Related to state estimation is the idea of parameter estimation.

Unlike a state, which we will define as a physical quantity that changes over time, a parameter is constant over time.

Position and orientation are states of a moving vehicle, while the resistance of a particular resistor in the electrical sub-system of a vehicle would be a parameter.

3.1 Common Techniques in State Estimation

3.1.1 Method of Least Squares

Gauss summarized the approach as follows:

The most probable value of an unknown parameter is that which minimizes the sum of squared errors between what we observe and what we expect.

To illustrate how this works, let's take a simple example. Suppose we are trying to measure the value in ohms of a simple resistor within the drive system of an autonomous vehicle. To do this, we grab a relatively inexpensive multi-meter that's lying around the lab.

Now, let's say we collect these four separate measurements, sequentially.

Measurement	Resistance (Ohms)
1	1068
2	988
3	1002
4	996

Measurement Model

'Actual' resistance *Measurement noise*

$$y_1 = x + v_1$$

$$y_2 = x + v_2$$

$$y_3 = x + v_3$$

$$y_4 = x + v_4$$

Fig.3.1: Measurement model of Resistance data

The type of carbon film resistors shown here is color-coded according to its rated resistance value. This resistor is rated at one 1kΩ. However, due to a number of factors, the true resistance can vary from the rated value. In this case, the resistor has a gold band which indicates that it can vary by as much as five percent. Furthermore, let's imagine that our multi-meter is particularly poor and that the person making the measurements is not particularly careful. For now, it's sufficient to treat this noise as equivalent to a general error. For each of the four measurements, we define a scalar noise term that is independent of the other noise terms.

Measurement Model	Squared Error
$y_1 = x + v_1$	$e_1^2 = (y_1 - x)^2$
$y_2 = x + v_2$	$e_2^2 = (y_2 - x)^2$
$y_3 = x + v_3$	$e_3^2 = (y_3 - x)^2$
$y_4 = x + v_4$	$e_4^2 = (y_4 - x)^2$

Fig. 3.2: Squared error model

In this case that the noise is independent and identically distributed or IID. Next, we define the error between each measurement and the actual value of our resistance x.

To find x, we square these errors to arrive at an equation that is a function of our measurements and the unknown resistance that we're looking for. With these errors defined, the method of

least squares says that the resistance value we are looking for, that is the best estimate of x is one which minimizes the squared error criterion, also sometimes called the squared error cost function or loss function.

To minimize the squared error criterion,

$$\hat{}_{LS} = \text{argmin}_x(e1^2 + e^2_2 + e^2_3 + e^2_4) = \quad ( )$$

We'll rewrite our errors in matrix notation. This will be especially helpful when we have to deal with hundreds or even thousands of measurements.

$$L_{LS}(x) = e_1^2 + e^2_2 + e^2_3 + e^2_4 = e^Te$$
$$= \quad y^Ty - x^TH^Ty - y^THx + x^TH^THx$$

Matrix H is called the Jacobian matrix. H has the dimensions m by n, where m is the number of measurements and n is the number of unknowns or parameters that we wish to estimate. From calculus, we know that we can do this by taking a derivative of the error function with respect to the unknown x and setting the derivative to zero.

$$\left|\frac{\partial L}{\partial x_{est}}\right. = -y^TH - y^TH + 2\hat{x}^TH^TH = 0$$
$$= -2y^TH + 2\hat{x}^TH^TH = 0$$

Re-arranging, we arrive at:

$$\hat{x}_{LS} = (H^TH)^{-1}H^Ty$$

x-hat minimizes our squared error criterion!

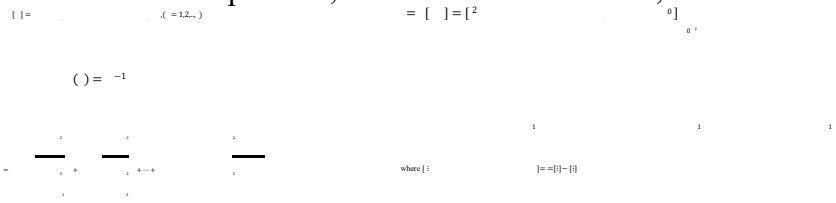
The matrix will be invertible if and only if m is greater than or equal to n. So, we need at least as many measurements as unknowns in order to derive the least squares solution.

3.1.2 Method of Weighted Least Squares

This method is preferred over least square method because some of our measurements are of better quality than others. We may want to trust certain measurements more than others as they may come from a better sensor.

For example, a number of our resistance measurements could have come from a much more expensive multi-meter than the others. Further, from now on, we'll also drop the assumption that we're only estimating one parameter and derive the more general normal equations. This will allow us to formulate a way to estimate multiple parameters at one time. If we wanted to estimate several resistance values at once. One way to interpret the ordinary method of least squares is to say that we are implicitly assuming that each noise term v_i is an independent random variable across measurements and has an equal variance or standard deviation.

If we assume each noise term is independent, but of different variance,



The higher the expected noise, the lower the weight we place on the measurement.

By expanding this expression, we can see why we call this weighted least square. Each squared error term is now weighted by the inverse of the variance associated with the corresponding measurement. In other words, the lower the variance of the noise, the more strongly it's associated error term will be weighted in the loss function.

We care more about errors which come from low noise measurements since those should tell us a lot about the true values of our unknown parameters.

The minimization is done in the same way as before, setting the derivative to zero.

Expanding our new criterion,

$$L_{WLS}(x) = e^T R^{-1} e$$
$$= (y - Hx)^T R^{-1} (y - Hx)$$

We can minimize it as before, but accounting for the new weighting term:

$$\hat{x} = \text{argmin}_x L(x) \rightarrow \left| \frac{\partial L}{\partial x} \right|_{x=\hat{x}} = 0 = -y^T R^{-1} H + \hat{x}^T H^T R^{-1} H$$

$$H^T R^{-1} H \hat{x}_{WLS} = H^T R^{-1} y$$

The weighted normal equations!

3.1.3 Method of Recursive Least Square

In the above methods, we assumed that we collected a batch of measurements and we wanted to use those measurements to compute our estimated quantities of interest.

This is sometimes a completely reasonable assumption. What can we do if instead, we have a stream of data? Do we need to recompute the least-squares solution every time we receive a new measurement?

For example, let's say we have a multi-meter that can measure resistance 10 times per second. Ideally, we'd like to use as many measurements as possible to get an accurate estimate of the resistance. If we use the method of least squares however, the number of computational resources we will need to solve our normal equations will grow with the measurement vector size. Alternatively, we can try and use a recursive method one that keeps a running estimate of the optimal parameter for all of the measurements that we've collected up to the previous time step and then updates that estimate given the measurement at the current time step. To do this we use a recursive algorithm, incrementally updating our estimate as we go along.

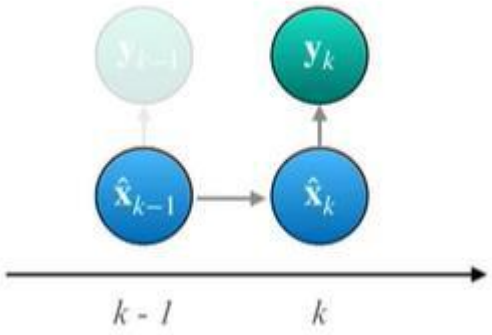


Fig. 3.3: State model

Let us assume that we have our best optimal estimate at time k-1. At time k we receive a new measurement that will, assume, follows linear measurement model with additive Gaussian noise. Our goal is to compute an updated optimal estimate at time k, given our measurement and the previous estimate.

A linear recursive estimate is given by the following expression:

$$\hat{} = \hat{}_{-} + \quad (- \quad \hat{}_{-})$$

Here k is called an estimator gain matrix. The term in brackets is called the innovation. It quantifies how well our current measurement matches our previous best estimate.

The new estimate is simply the sum of the old estimate and corrective term based on the difference between what we expected the measurement to be and what we actually measured. In fact, if the innovation were equal to zero, we would not change our old estimate at all.

Now to compute k, we'll need to use a recursive least-squares criterion and some matrix calculus as before. For a single scalar parameter like resistance, this amounts to minimizing the estimator state variance. For multiple unknown parameters, this is equivalent to minimizing the trace of state covariance matrix at time t. This is exactly like our former least-squares criterion except now we have to talk about expectations.

Instead of minimizing the error directly, we minimize its expected value which is actually the estimator variance. The lower the variance, the more we are certain of our estimate.

The least-squares algorithm looks like this-

1. Initialize the estimator

$$\hat{x}_0 = E[x]$$
$$P_0 = E[(x - \hat{x}_0)(x - \hat{x}_0)^T$$

2. Set up the measurement model, defining the Jacobian and the measurement covariance matrix:

$$y_k = H_k x + v_k$$

3. Update the estimate of ˆ and the covariance using:

$$K_k = P_{k-1} H_k^T (H_k P_{k-1} H_k^T + R_k)^{-1}$$
$$\hat{x}_k = \hat{x}_{k-1} + K_k(y_k - H_k \hat{x}_{k-1})$$
$$P_k = (1 - K_k H_k) P_{k-1}$$

We initialize the algorithm with an estimate of our unknown parameters and a corresponding covariance matrix. This initial guess could come from the first measurement we take and the covariance could come from technical specifications. Next, we set up our measurement model and pick values for our measurement covariance. Finally, every time a measurement is recorded, we compute the measurement gain and then use it to update our estimate of the parameters and our estimator covariance or uncertainty.

Recursive least square enables us to minimize computational effort in our estimation process which is always a good thing. More importantly, recursive least squares form the update step of the linear Kalman filter.

3.1.4 Linear Kalman Filter

History: The Kalman filter algorithm was published in 1960 by Rudolf E. Kalman, a Hungarian born professor and engineer who was working at the Research Institute for Advanced Studies in Baltimore Maryland. Years later in 2009, American President Barack Obama awarded Kalman the prestigious National Medal of Science for his work on the Kalman filter and other contributions to the field of control engineering. After its publication in 1960, the Kalman filter was adopted by NASA for use in the Apollo guidance computer. It was this ground-breaking innovation that played a pivotal role in successfully getting the Apollo spacecraft to the moon, and to our first steps on another world. The filter helped guide the Apollo spacecraft accurately through its circumlunar orbit. The engineers at NASA's Ames Research Center, adopted Kalman's linear theory and extended it to nonlinear models.

Basic concept: The Kalman filter is very similar to the linear recursive least squares filter. While recursive least squares update the estimate of a static parameter, but Kalman filter is able to update and estimate of an evolving state. The goal of the Kalman filter is to take a probabilistic estimate of this state and update it in real-time using two steps; prediction and correction. To make these ideas more concrete, let's consider a problem of estimating the 1D position of the vehicle.

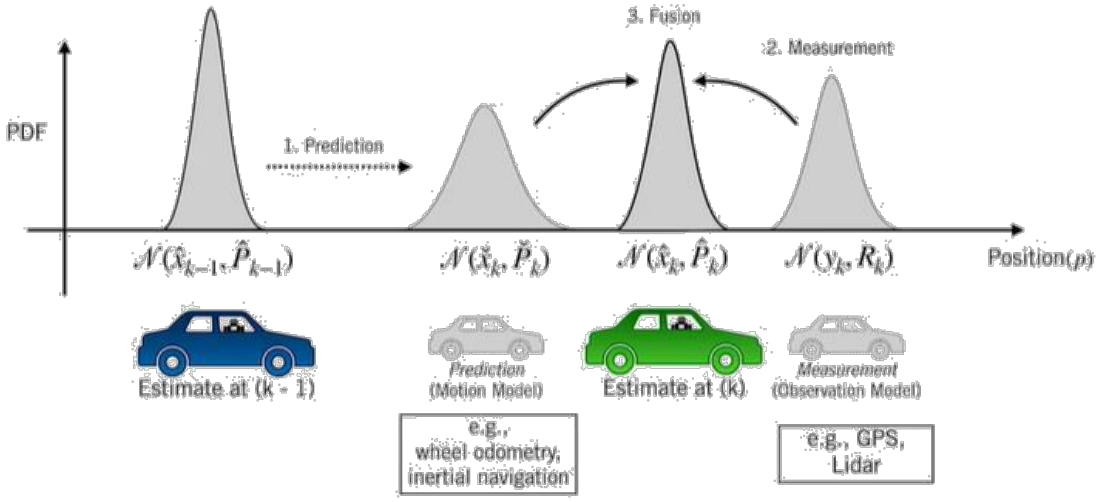


Fig. 3.4: Prediction and Correction method

Starting from an initial probabilistic estimate at time (k-1), our goal is to use a motion model which could be derived from wheel odometry or inertial sensor measurements to predict our new state. Then, we'll use the observation model derived from GPS for example, to correct that prediction of vehicle position at time k. Each of these components, the initial estimate, the predicted state, and the final corrected state are all random variables that we will specify by their means and covariances. In this way, we can think of the Kalman filter as a technique to fuse information from different sensors to produce a final estimate of some unknown state, taking into account, uncertainty in motion and in our measurements.
For the Kalman filter algorithm, we had been able to write the motion model and measurement model in the following way;

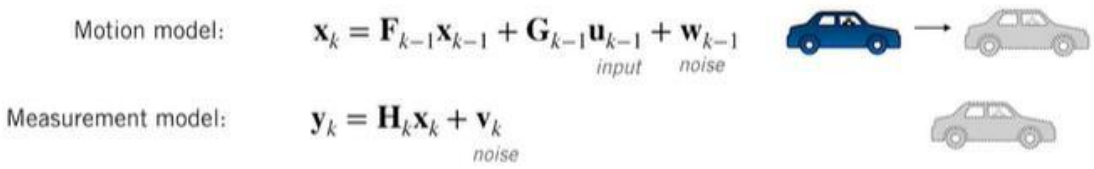


Fig. 3.5: Motion and Measurement model

We can use an approach very similar to that we discussed in the recursive least squares video. Except this time, we'll do it in two steps. First, we use the process model to predict how our states, remember, that we're now typically talking about evolving states and non-state parameters evolved since the last time step, and will propagate our uncertainty. Second, we'll use our measurement to correct that prediction based on our measurement residual or innovation and our optimal gain. Finally, we'll use the gain to also propagate the state covariance from our prediction to our corrected estimate.

1. Prediction

$\hat{x}_k = F_k \hat{x}_{k-1} + G_k u_{k-1}$

R_k - given motion model at time k

2a. Optimal Gain

$K_k = \hat{P}_k H_k^T (H_k \hat{P}_k H_k^T + R_k)^{-1}$

$\check{x}_k$ - given measurement at time k

2b. Correction

(− ˇ) is often called the innovation

In our notation, the hat indicates a corrected prediction at a particular time step. Whereas a check indicates a prediction before the measurement is incorporated. We start with a probability density over our states and also maybe parameters at time step k-1, which we represent as a multivariate Gaussian. We then predict the states at time step k using our linear prediction model and propagate both the mean and the uncertainty; the covariance, forward in time. Finally, using our probabilistic measurement model, we correct our initial prediction by optimally fusing the information from our measurement together with the prior prediction through our optimal gain matrix k. Our end result is an updated probabilistic estimate for our states at time step k. The best way to become comfortable with the Kalman filter is to use it. Let's look at a simple example. Consider again the case of the self-driving vehicle estimating its own position. Our state vector will include the vehicle position and its first derivative velocity. Our input will be a scalar acceleration that could come from a control system that commands our car to accelerate forward or backward. For our measurement, we'll assume that we're able to determine the vehicle position directly using something like a GPS receiver.

3.1.5 Extended Kalman Filter

Linear systems don't exist in reality. Even a very simple system like a resistor with a voltage applied isn't truly linear, at least not all the time. For a certain range of voltages, the current is a linear function of the voltage and follows Ohm's law. But as the voltage gets higher, the resistor heats up, which alters the resistance in a non-linear way.

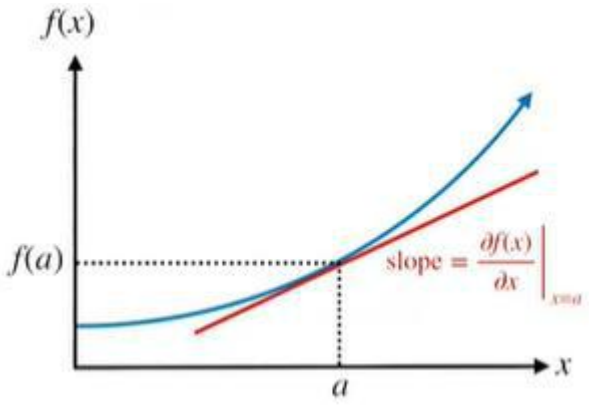


Fig. 3.6: Linearization of non-linearized functiom

The key concept in the extended Kalman filter is the idea of linearizing a non-linear system. For this reason, the EKF is sometimes referred to as the linearized Kalman filter.

Linearizing a system just means picking some operating point a, and finding a linear approximation to the nonlinear function in the neighbourhood of a. In two dimensions, this means finding the tangent line to the function f of x, when x equals a. Mathematically, we do this by taking the Taylor series expansion of the function. Taylor series expansion is a way of representing a function as an infinite possibility, some whose terms are calculated from the function's derivatives at a single point. For linearization, we're only interested in the first-order terms of the Taylor series expansion highlighted in red.

$$f(x) \approx f(a) + \left|\frac{df(a)}{dx}\right| (x - a)$$

First order term Higher order terms

Ideally, we would like to linearize the models about the true value of the state, but we can't do that because we already knew the true value of the state, we wouldn't need to estimate it. So instead, let's pick the next best thing: the most recent estimate of the state. For our motion model, we'll linearize about the posterior estimate of the previous state, and for the measurement model, we'll linearize about our prediction of the current state based on the motion model.

Linearized motion model

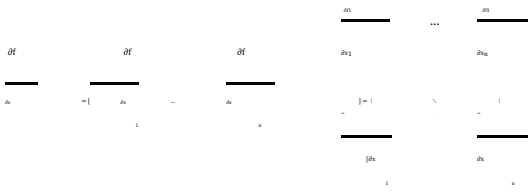
$$x_k = f_{k-1}(x_{k-1}, u_{k-1}, w_{k-1})$$

F_{k-1} L_{k-1}

Linearized measurement model

$$y = h(x, v) \approx h(\check{x}, 0) + \left|\frac{\partial h_k}{\partial x}\right| (x - \check{x}) + \left|\frac{\partial h_k}{\partial v}\right| v$$

So now we have a linear system in state-space, and the matrices F, L, H, and M are called Jacobian matrices of the system.



Computing these matrices correctly is the most important and difficult step in the extended Kalman filter algorithm, and it's also the most commonplace to make mistakes. In vector calculus, a Jacobian or Jacobian matrix is the matrix of all first-order partial derivatives of a vector-valued function. Each column of the Jacobian contains the derivatives of the function

outputs with respect to a given input. For example, if your function takes a three-dimensional vector and spits out a two-dimensional vector, the Jacobian would be a two by three matrix. Intuitively, the Jacobian matrix tells you how fast each output of your function is changing along each input dimension. Just like how the derivative of a scalar function tells you how fast the output is changing as you vary the input. The Jacobian is really just a generalization of the first derivative to multiple dimensions.

Now, we know how to compute the Jacobian matrices needed for the EKF, and all that's left is to plug them into our standard Kalman filter equations. There are a couple of differences to notice in the EKF equations compared to the Kalman filter equations. First, in the prediction and correction steps, we're still using the non-linear models to propagate the mean of the state estimate and to compute the measurement residual or innovation. That's because we linearized our motion model about the previous state estimate, and we linearize the measurement model about the predicted state. By definition, the linearized model exactly coincides with the non-linear model at the operating point. The second difference is the appearance of the L and M Jacobians related to the process and measurement noise. In many cases, both of these matrices will be identified since noise is often assumed to be additive, but this is not always the case.

Linearized motion model

$$x_k = f_{k-1}(\hat{x}_{k-1}, u_{k-1}, 0) + F_{k-1}(x_{k-1} - \hat{x}_{k-1}) + L_{k-1}w_{k-1}$$

Linearized measurement model

$$y_k = h_k(\check{x}_k, 0) + H_k(x_k - \check{x}_k) + M_k v_k$$

Prediction

$$\check{x}_k = f_{k-1}(\hat{x}_{k-1}, u_{k-1}, 0)$$

$$\check{P}_k = F_{k-1}\hat{P}_{k-1}F_{k-1}^T + L_{k-1}Q_{k-1}L_{k-1}^T$$

f_k – given motion model at time k

h_k – given measurement model at time k

Optimal Gain

$$K_k = \check{P}_k H_k^T (H_k \check{P}_k H_k^T + M_k R_k M_k^T)^{-1}$$

Correction

$$\hat{x}_k = \check{x}_k + K_k(y_k - h_k(\check{x}_k, 0))$$

$$\hat{P}_k = (1 - K_k H_k)\check{P}_k$$

3.2 Localization using ROS (Robot Operating System)

SLAM (simultaneous localization and mapping) is a technique for creating a map of environment and determining robot position at the same time. It is widely used in robotics.

While moving, current measurements and localization are changing, in order to create map it is necessary to merge measurements from previous positions.

For definition of SLAM problem we use given values (1A,2) and expected values (3,4):

1. Robot control

$_{1:}$ $= \{u1, u2, u3, \ldots, ut\}$

2. Observations

$z_{\{1:t\}} = \{z1, z2, z3, \ldots, \}$

3. Environment map

m

4. Robot trajectory

$x_{\{1:t\}} = \{x1, x2, x3, \ldots, xt\}$

6. Robot trajectory estimates are determined with:

$p(x_{0:t}, m | z_{1:t}, u_{1:t})$

3.2.1 Coordinate Frame Tracking

ROS can help us with keeping track of coordinate frames over time. Package for it is tf2 - the transform library. A robotic system typically has many 3D coordinate frames that change over time, such as a world frame, base frame, gripper frame, head frame, etc. tf2 keeps track of all these frames over time, and allows you to ask questions like:

- Where was the head frame relative to the world frame, 5 seconds ago?
- What is the pose of the object in my gripper relative to my base?
- What is the current pose of the base frame in the map frame?

tf2 can operate in a distributed system. This means all the information about the coordinate frames of a robot is available to all ROS components on any computer in the system. Tf2 can operate with a central server that contains all transform information, or you can have every component in your distributed system build its own transform information database.

tf2 package comes with a specific message type: tf/Transform and it is always bound to one topic: /tf. Message tf/Transform consist of transformation (translation and rotation) between two coordinate frames, names of both frames and timestamp.

3.2.1.1 Publishing Transformation

Publishing to /tf is done in different way than to any other topic, we will write tf publisher in the example. We will make a node that subscribe to /pose topic with message type geometry_msgs/PoseStamped and publish transformation between objects mentioned in /pose. This node is required only when we are working with hardware, Gazebo is publishing necessary tf frames itself.

Final code for publishing to /tf:

```
#include <ros/ros.h>
#include <geometry_msgs/PoseStamped.h>
#include <tf/transform_broadcaster.h>

tf::Transform transform;
tf::Quaternion q;

void pose_callback(const_geometry_msgs::PoseStampedPtr &pose)

{

    Static tf::TransformBroadcaster br;
    q.setX(pose->pose.orientation.x);
    q.setY(pose->pose.orientation.y);
    q.setZ(pose->pose.orientation.z);
    q.setW(pose->pose.orientation.w);

    transform.setOrigin(tf::Vector3(pose.position.x, pose->pose.position.y, 0.0));
    transform.setRotation(q);

    br.sendTransform(tf::StampedTransform(transform, ros::Time::now(),
    "odom","base_link"));

}

int main (int argc, char **argv)

{

    ros::init(argc, argv, "drive_controller");
    ros::NodeHandle n("~");
    ros::Subscriber pose_sub = n.subscribe("/pose", 1, pose_callback);
    ros::Rate loop_rate(100);
    while (ros::ok())

    {

        ros::spinOnce();
        loop rate.sleep();

    }

}
```

Code Explanation:

- Begin with headers:

  #include <ros/ros.h>
  #include <geometry_msgs/PoseStamped.h>
  #include <tf/transform_broadcaster.h>

- Publisher for transformation:

  Static tf::TransformBroadcaster br;

- Transformation message:

  tf::Transform transform;

- Quaternion for storing rotation data:

  tf::Quaternion q;

- Function for handling incoming /PoseStamped messages, extract quaternion parameters from message and put it to quaternion structure, put translation parameters into transform message, put quaternion structure into transform message, publish transform: *void pose_callback(const_geometry_msgs::PoseStampedPtr &pose)*

  {

  Static tf::TransformBroadcaster br;
  q.setX(pose->pose.orientation.x);
  q.setY(pose->pose.orientation.y);
  q.setZ(pose->pose.orientation.z);
  q.setW(pose->pose.orientation.w);

  transform.setOrigin(tf::Vector3(pose.position.x, pose->pose.position.y, 0.0)); transform.setRotation(q);

  br.sendTransform(tf::StampedTransform(transform, ros::Time::now(), "odom","base_link"));

  }

- Publishing of transform is done with send Transform function which parameter is StampedTransform object. This object parameters are:
  - transform - transformation data
  - ros::Time::now() - timestamp for current transformation
  - odom - transformation parent frame - the one that is static
  - base_link - transformation child frame - the one that is transformed

- In main function just initialize node and subscribe to /pose topic:

```
int main (int argc, char **argv)
{
    ros::init(argc, argv, "drive_controller");
    ros::NodeHandle n("~");
    ros::Subscriber pose_sub = n.subscribe("/pose", 1, pose_callback);
    ros::Rate loop_rate(100);
    while (ros::ok())
    {
        ros::spinOnce();
        loop rate.sleep();
    }
}
```

Save it as drive_controller.cpp and declare executables and specify libraries to CMakeLists.txt file.

In CMakeLists.txt file find line:

```
find_package(catkin REQUIRED COMPONENTS
roscpp
)
```

and change it to:

```
find_package(catkin REQUIRED COMPONENTS
roscpp tf
)
```

then declare executable:

```
add_executable(drive_controller_node src/drive_controller.cpp)
```

And specify libraries to link:

```
Target_link_libraries(drive_controller_node
${catkin_LIBRARIES}
)
```

3.2.1.2 Publisher for static transformations

If we want to publish transformation which is not changing in time, we can use static_transform_publisher from tf package. We will use it for publishing relation between robot base and laser scanner.

We can run it from command line:

rosrun tf static_transform_publisher 0 0 0 3.14 0 0 base_link laser 100

Arguments are consecutively:

- 0 0 0 - x y z axes of translation, values are in meters
- 3.14 0 0 - z y x axes of rotation, values are in radians
- base_link - transformation parent frame - the one that is static
- laser - transformation child frame - the one that is transformed
- 100 - delay between consecutive messages

You can use it to adjust position of your laser scanner relative to robot. The best would be, if you place scanner in such a way that its rotation axis is coaxial with robot rotation axis and front of laser scanner base should face the same direction as robot front. Most probably your laser scanner will be attached above robot base. To set scanner 10 centimeters above robot we should use:

rosrun tf static_transform_publisher 0 0 0.1 0 0 0 base_link laser 100

And if your scanner is also rotated by 180º around z-axis it should be:

rosrun tf static_transform_publisher 0 0 0.1 3.14 0 0 base_link laser 100

Remember that if we have improperly mounted scanner or its position is not set correctly, our map will be generated with errors or it will be not generated at all.

3.2.1.3 Visualizing transformation with rviz

- We can visualize all transformations that are published in the system. To test it run:

  roslaunch rosbot_gazebo maze_world.launch -Start Gazebo with a maze build on it

- We will also need rviz and keyboard controller:

  teleop_twist_keyboard - keyboard controller
  rviz - visualization tool

- We may also add some static_transform_publisher nodes.

So, we will launch the file:

```
<launch>

    <include file="$(find rosbot_gazebo)/launch/maze_world.launch" />
    <include file="$(find rosbot_description)/launch/rosbot_gazebo.launch"/>
    <param name="use_sim_time" value="true" />

<node pkg="tf" type="static_transform_publisher" name="laser_broadcaster"
args="0 0 0 3.14 0 0 base_link laser 100" />
<node pkg="tf" type="static_transform_publisher" name="camera_publisher"
args="-0.03 0 0.18 0 0 0 base_link camera_link 100" />

<node pkg="rviz" type="rviz" name="rviz" args="-d $(find
tutorial_pkg)/rviz/tutorial_7.rviz"/>

</launch>
```

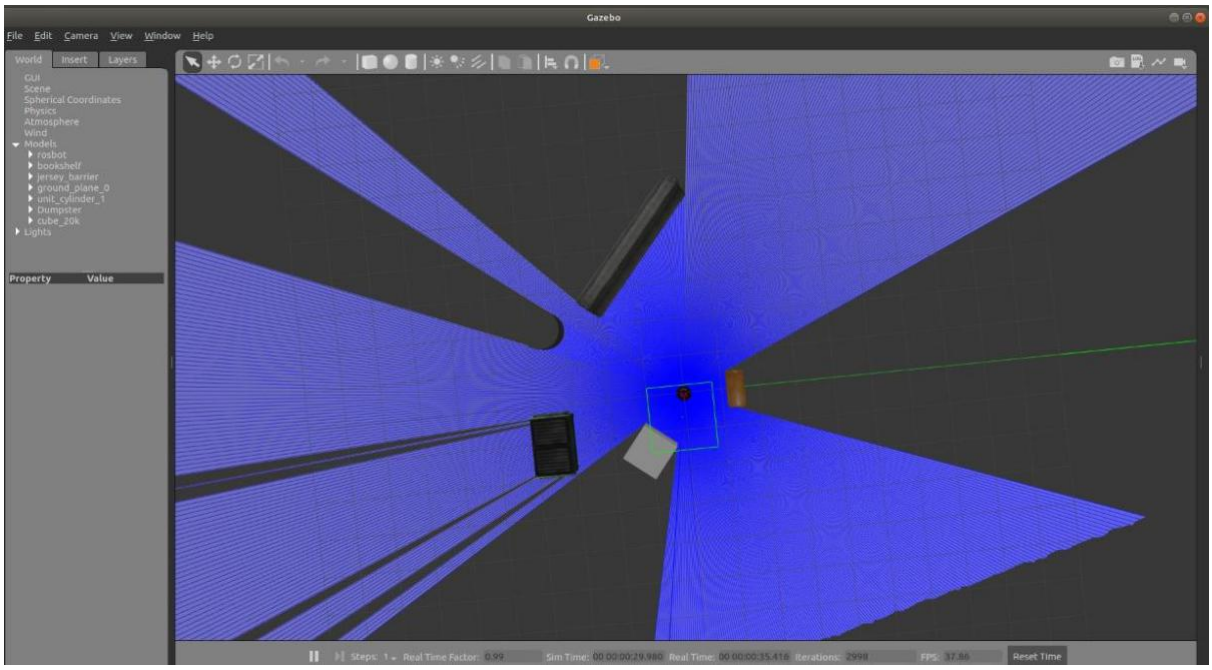


Fig. 3.7: Gazebo Window (bird eye view)

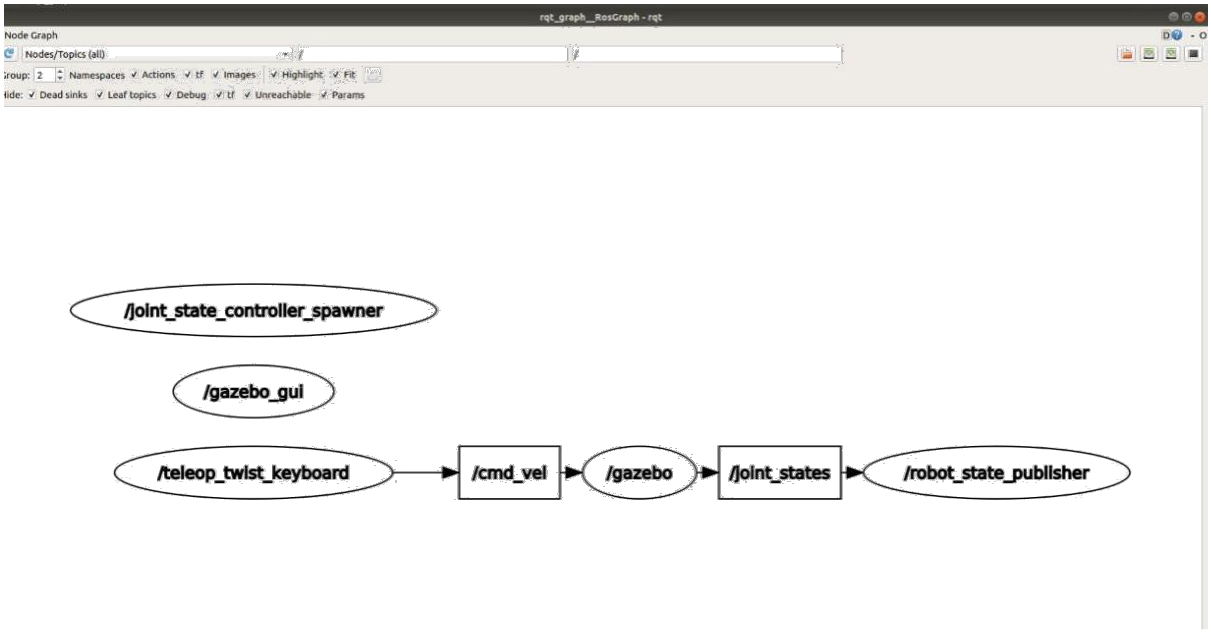


Fig. 3.8: rqt_graph

A Gazebo and a rviz window will open. Now click “Add” from object manipulation buttons, in new window select “By display type” and from the list select “Tf”. We can also add “Pose” visualization. We can observe as robot_base frame moves accordingly to robot movement.

3.2.2 Running the Laser Scanner

In hardware, we will use “rpLidar” laser scanner. Place it on your robot, main rotation axis should pass the centre of robot. Front of the “rpLidar” should face the same direction as front of the robot.
As a driver for the “rpLidar” we will use rplidarNode from rplidar_ros package. This node communicate with device and publish its scans to /scan topic with message type sensor_msgs/LaserScan. We do not need more configuration for it now. To test it you can run only this one node:

rosrun rpLidar ros rplidarNode

For Gazebo we do not need any additional nodes, just start simulator and laser scans will be already published to appropriate topic.
We have /scan topic in your system. We can examine it with rostopic info but better do not try to echo it, it is possible but we will get lots of output that is hard to read. Better method for checking the /scan topic is to use rviz. Run rviz and click “Add” from object manipulation buttons, in new window select “By topic” and from the list select /scan. In visualized items list find position Fixed Frame and change it to laser. To improve visibility of scanned shape, we

may need to adjust one of visualized object options, set value of Style to Points. We should see many points which resemble shape of obstacles surrounding our robot

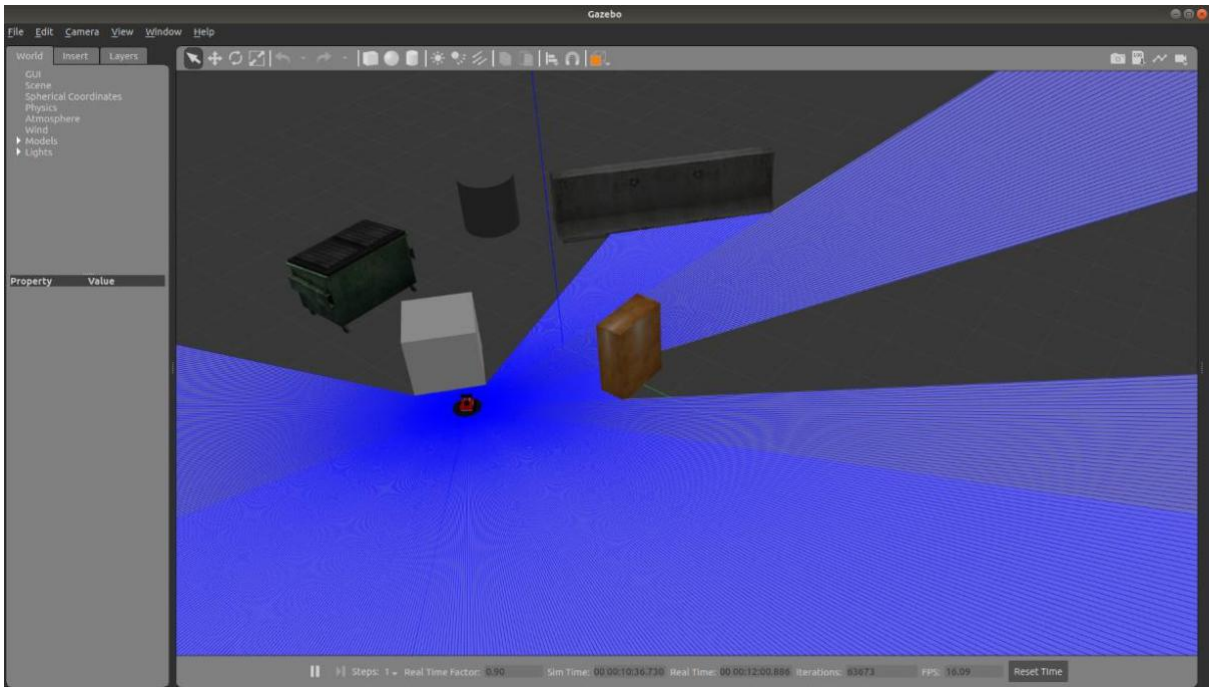


Fig. 3.9: Gazebo Window (side-view)

In rviz add Tf and /scan. This time set Fixed Frame to odom.

Try to move around the robot, we'll see as laser scans change its shape accordingly to obstacles passed by robot.

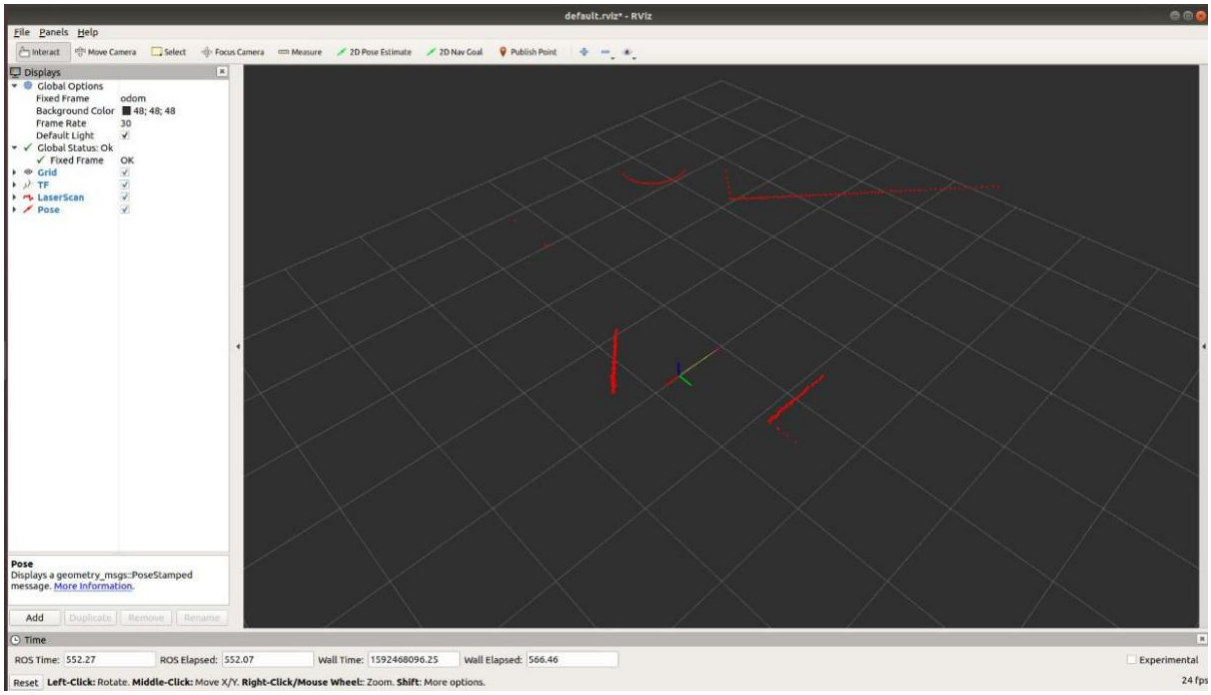


Fig. 3.10: rviz Win. showing Laser scans

Chapter-4

NAVIGATION AND MAP BUILDING

For building a map and localizing robot relative to it, we will use slam_gmapping node from gmapping package.

The gmapping package provides laser-based SLAM (Simultaneous Localization and Mapping), as a ROS node called slam_gmapping. Using slam_gmapping, we can create a 2-D occupancy grid map (like a building floorplan) from laser and pose data collected by a mobile robot.
The slam_gmapping node takes in sensor_msgs/LaserScan messages and builds a map (nav_msgs/OccupancyGrid). The map can be retrieved via a ROS topic or service.

4.1 Message: sensor_msgs/Laserscan

File: sensor_msgs/LaserScan.msg

4.1.1 Raw Message Definition:

- Header header: timestamp in the header is the acquisition time of the first ray in the scan. In frame frame_id, angles are measured around the positive z-axis (counter clockwise, if z is up) with zero angle being forward along the x-axis.
- float32 angle_min: start angle of the scan [rad]
- float32 angle_max: end angle of the scan [rad]
- float32 angle_increament: angular distance between measurements [rad]
- float32 time_increament: time between measurements [seconds] – if scanner is moving, this will be used in interpolating position of 3-d points.
- float32 scan_time: time between scans [seconds]
- float32 range_min: minimum range value [m]
- float32 range_max: maximum range value [m]
- float32[] ranges: range data [m] (Note; values<range_min or > range_max should be discarded)
- float32[] intensities: intensity data [device-specific units]. If device does not provide intensities, leave the array empty.

4.1.2 Compact Message Definition

- 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

4.2 Message: nav_msgs/OccupancyGrid

File: nav_msgs/OccupancyGrid.msg

4.2.1 Raw Message Definition

This represents a 2-D grid map, in which each cell represents the probability of occupancy.

- Header header
- MapMetaData info: MetaData for the map
- int8[] data: The map data, in row-major order, starting with (0,0). Occupancy probabilities are in range [0,100]. Unknown is -1.

4.2.2 Compact Message Definition

- std_msgs/Header header
- nav_msgs/MapMetaData info
- int8[] data

This node subscribes /tf topic to obtain robot pose relative to starting point and laser scanner pose relative to robot and also subscribe /scan topic to obtain laser scanner messages. Node publishes to /map topic with message type nav_msgs/OccupancyGrid, this contain the actual map data.
We need to set few parameters:

- base_frame - name of frame related with robot, in our case it will be base_link
- odom_frame - name of frame related with starting point, in our case it will be odom
- delta - map resolution expressed as size of every pixel in meters

We can use below launch file:

```
<launch>

   <include file="$(find rosbot_gazebo)/launch/maze_world.launch" />
   <include file="$(find rosbot_description)/launch/rosbot_gazebo.launch"/>
   <param name="use_sim_time" value="true" />

   <node pkg="tf" type="static_transform_publisher" name="laser_broadcaster" args="0
   0 0 3.14 0 0 base_link laser 100" />
   <node pkg="tf" type="static_transform_publisher" name="camera_publisher" args="-
   0.03 0 0.18 0 0 0 base_link camera_link 100" />

   <node pkg="gmapping" type="slam_gmapping" name="gmapping_node" output="log">
      <param name="base_frame" value="base_link" />
      <param name="odom_frame" value="odom" />
      <param name="delta" value="0.01" /> <param
      name="xmin" value="-5" /> <param
      name="ymin" value="-5" /> <param
      name="xmax" value="5" /> <param
      name="ymax" value="5" />
      <param name="maxUrange" value="5" />
      <param name="map_update_interval" value="1" />
      <param name="linearUpdate" value="0.05" />
      <param name="angularUpdate" value="0.05" />
      <param name="temporalUpdate" value="0.1" />
      <param name="particles" value="100" />
   </node>

   <node pkg="rviz" type="rviz" name="rviz" args="-d $(find
    tutorial_pkg)/rviz/tutorial_6.rviz"/>

</launch>
```

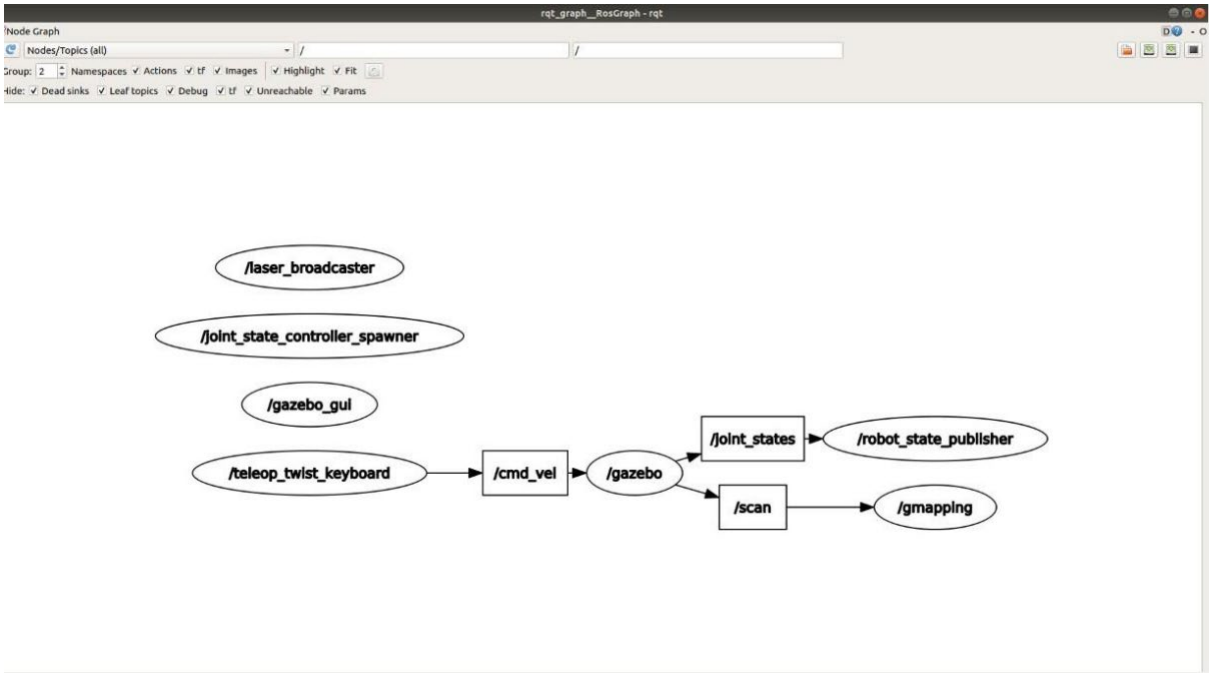


Fig. 4.1: rqt_graph

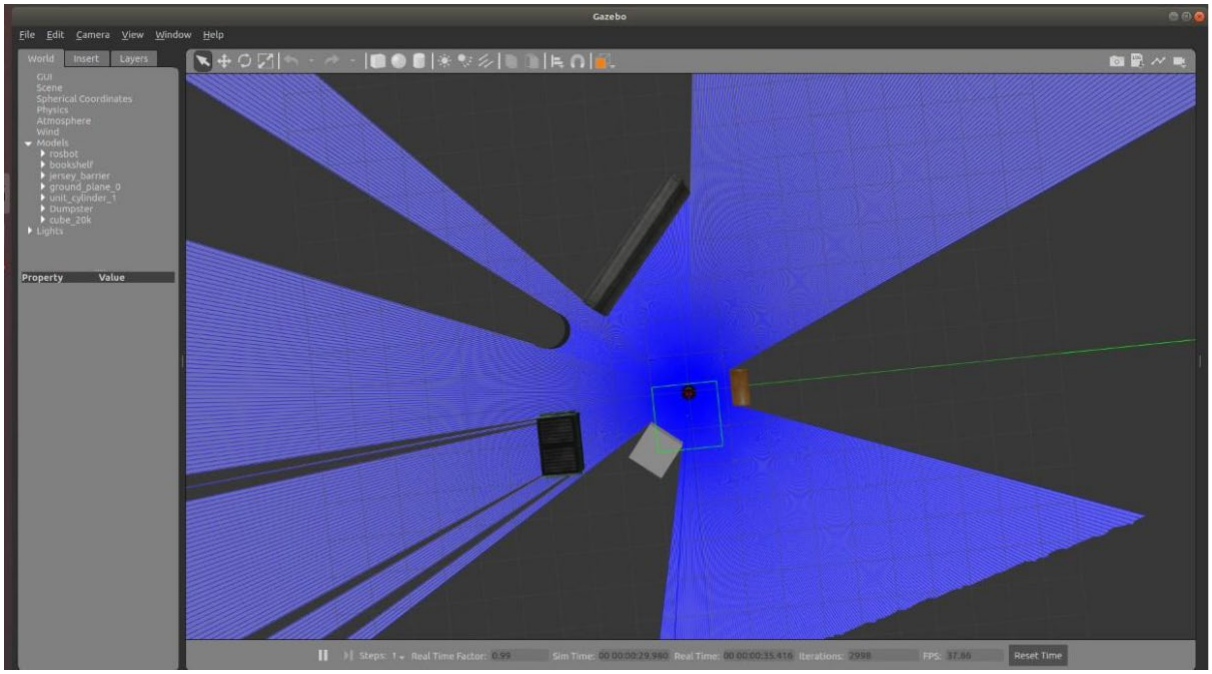


Fig. 4.2: Gazebo Window

In rviz add Tf and /scan, again open object adding window, select “By topic” and from the list select /map.

At the beginning there could be no map, we may need to wait few second until it is generated. Starting state should be similar to the one on picture:

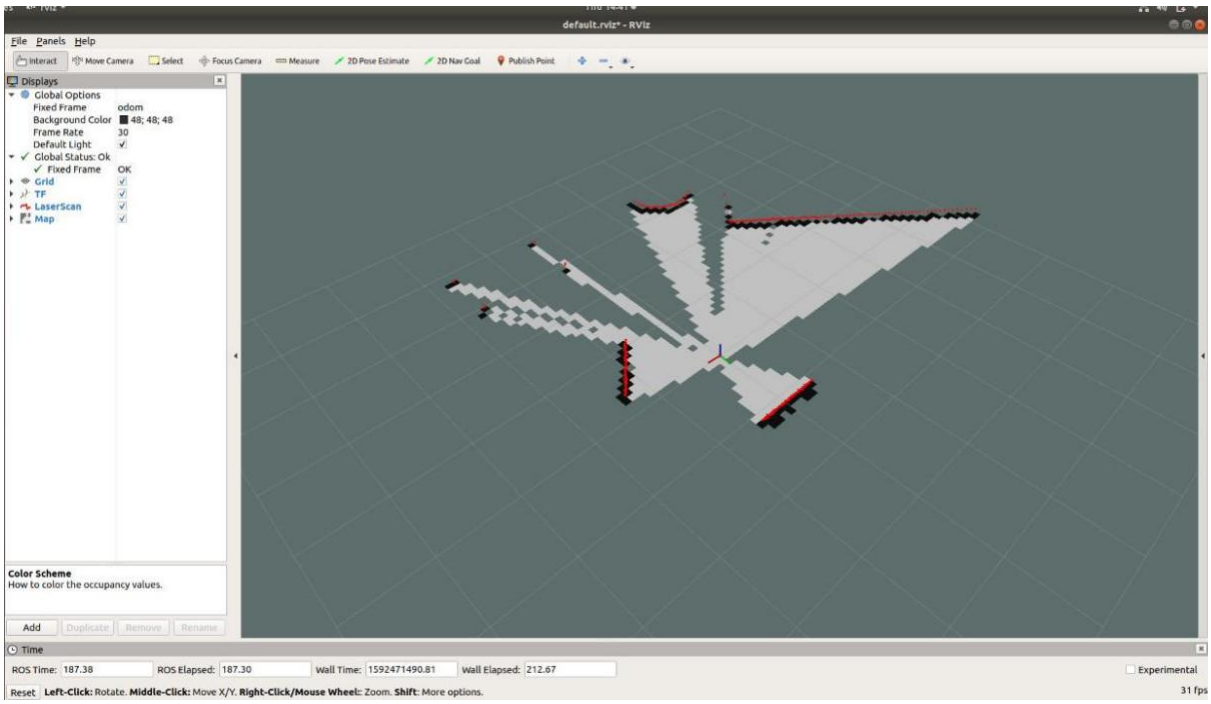


Fig. 4.3: Map building in rviz window

Chapter-5

PATH PLANNING

Task of path planning for mobile robot is to determine sequence of manoeuvres to be taken by robot in order to move from starting point to destination avoiding collision with obstacles.

Path planning algorithms may be based on graph or occupancy grid.

5.1 Graph Methods

Method that is using graphs, defines places where robot can be and possibilities to traverse between these places. In this representation graph vertices define places e.g. rooms in building while edges define paths between them e.g. doors connecting rooms. Moreover, each edge can have assigned weights representing difficulty of traversing path e.g. door width or energy required to open it. Finding the trajectory is based on finding the shortest path between two vertices while one of them is robot current position and second is destination.

5.1.1 Dijkstra's Algorithm

Dijkstra's algorithm (or Dijkstra's Shortest Path First algorithm, SPF algorithm) is an algorithm for finding the shortest paths between nodes in a graph, which may represent, for example, road networks.

Dijkstra's algorithm to find the shortest path between a and b. It picks the unvisited vertex with the lowest distance, calculates the distance through it to each unvisited neighbour, and updates the neighbour's distance if smaller. Mark visited (set to red) when done with neighbours.

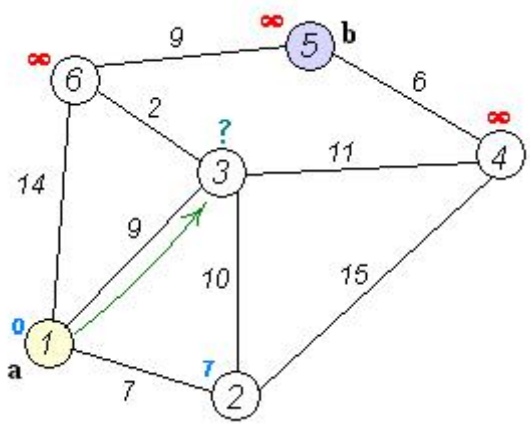


Fig. 5.1: Dijkstra's algorithm

For every searched node n, the D* algorithm computes the cost value g(n) of the optimal path from the node n to the goal node. The algorithm stores the backpointers for every searched node n, which point to the parent node with the smallest cost g. A backpointer is noted by the function b(·), where b(n) = m means that the node n has the smallest cost because it follows the node m. The backpointers ensure that optimal path from any searched node n to the goal node can be extracted according to the function b(·).

The D* algorithm starts the search from the goal node, examining neighbour nodes of minimal g value until the start node is reached. For better re-planning performances, the exhaustive search can be done in the initial phase, which computes optimal paths and path costs g from every reachable node in the graph G(N , E, W) to the goal node. If during the motion robot detects change of occupancy values, first the weights in the graph G(N , E, W) are updated. The backpointers are redirected locally and the new optimal path from the robot's current position is determined. The number of expanded nodes is minimal and consequently the time of execution. A simple illustration of the algorithm can be seen in Fig. 9.

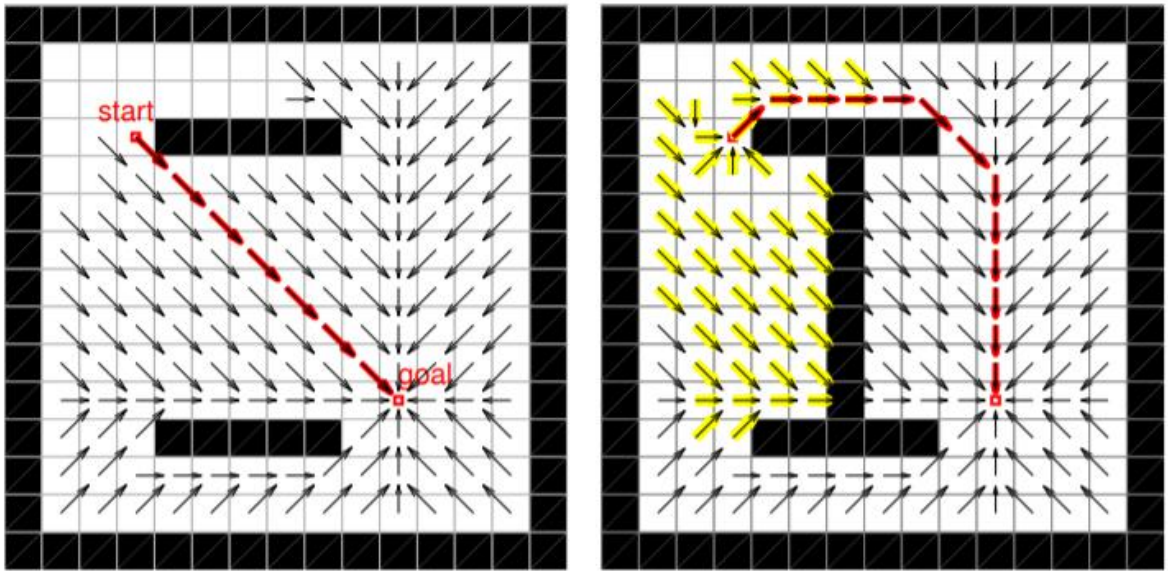


Fig. 5.2: Illustration of path planning(left) and re-planning(right) by Dijkstra's algorithm

5.1.2 A* Algorithm

A* is a graph traversal and path search algorithm, which is often used in computer science due to its completeness, optimality, and optimal efficiency.

A* is an informed search algorithm, or a best-first search, meaning that it is formulated in terms of weighted graphs: starting from a specific starting node of a graph, it aims to find a path to the given goal node having the smallest cost (least distance travelled, shortest time, etc.). It does this by maintaining a tree of paths originating at the start node and extending those paths one edge at a time until its termination criterion is satisfied.

An example of an A* algorithm in action where nodes are cities connected with roads and h(x) is the straight-line distance to target point:

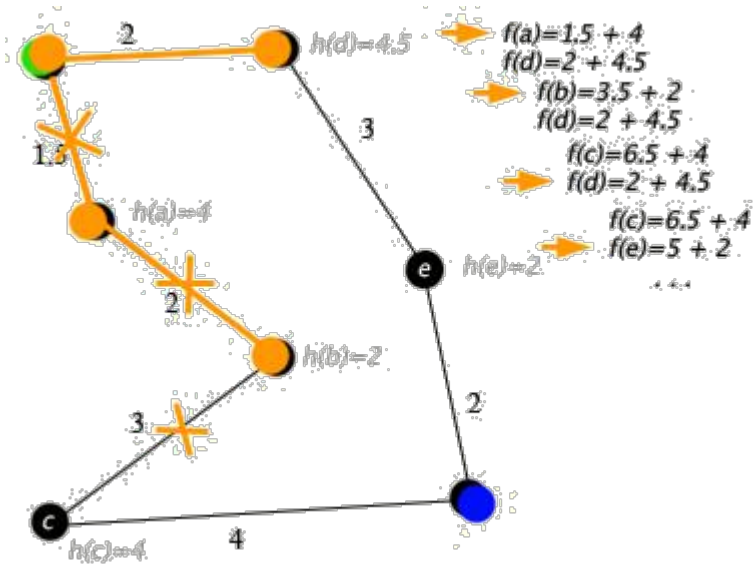


Fig 5.3: A* algorithm

A* algorithm is an improved heuristic searching strategy which has some restrictions on its evaluation function based on A method.

The key issue of A* algorithm is the designing of the cost estimation function f*(n) of the nodes. Each time the algorithm expands the node with the least cost value of the feasible nodes. Function g(n) expresses the real cost value from the origin node to node n. Function h*(n) expresses the least cost value from node n to the goal point, and it is unknown. So:

$$f^*(n)=g(n)+h^*(n)$$

Function f*(n) shows the least cost value from the origin node to the goal point while passing node n. In the global path planning for the mobile robot, the cost value f*(n) represents the number of the traversed grids by the robot, from the starting point to the goal point while passing node n. Every time the robot covers one grid, the exact value of g(n) can be calculated as the cumulative number of grids from the starting point to the current node n, while the estimation cost value is the cumulative number of grids from the current node to the goal point. When the robot wants to move, it has to judge the next aimed grid is the free one or not, then estimates the related attribute value of it. If the grid has the largest value of the related attribute, it can be selected as the next covering grid. If there are several grids having the same related values, the robot will resort to the grids which locate at the top, upper right, or right directions with regard to the start point to the goal point along S→G, and select a proper one.

The detailed algorithm is following:

1. Set a OPEN table to store the nodes needing to be expanded, and a CLOSED table to keep the already expanded nodes. At the beginning, the OPEN table saves all the visible external vertexes selected by the improved V-graph method, while the CLOSED table is empty. S represents the starting point, and G represents the target point. Set the estimation function h*(n)=dis(n,G), and the searching direction is along the straight line, the start point to the goal point S→G. The initial value of cell (x,y) visited is 0, and cell (x,y) related can be given a random positive integer at beginning;

2. Judge whether the OPEN table is empty or not. If yes, turn to step 8), otherwise continue;
3. Update the cost function of the expanded node n:

$$f^*(n) = \text{cell}(x,y).\text{visited} + h^*(n)$$

4. Judge whether the current point n is the aimed node which has the least cost value of f*(n) or not; If yes, turn to step 6), otherwise continue;
5. Node n is removed from the OPEN table and added to the CLOSED table;
6. The robot continues to search the next aimed node, along the direction from the start point to the goal point, and determine whether the cell (x,y) related value of the searched node is the largest or not. If yes, update the cell (x,y) visited value and the cell (x,y) related value based on 2) and 3), and go to step 5), otherwise continue;
7. The robot resorts to the upper, upper right, or right of the free grids based on the eight-connected relation around it and the direction of S→G. Update the cell (x,y) visited value and the cell (x,y) related value according to 2) and 3). Then the next aimed node n is found, and go to step 3);
8. The procedure ends. The algorithm flow chart is shown in Fig. 5.4.

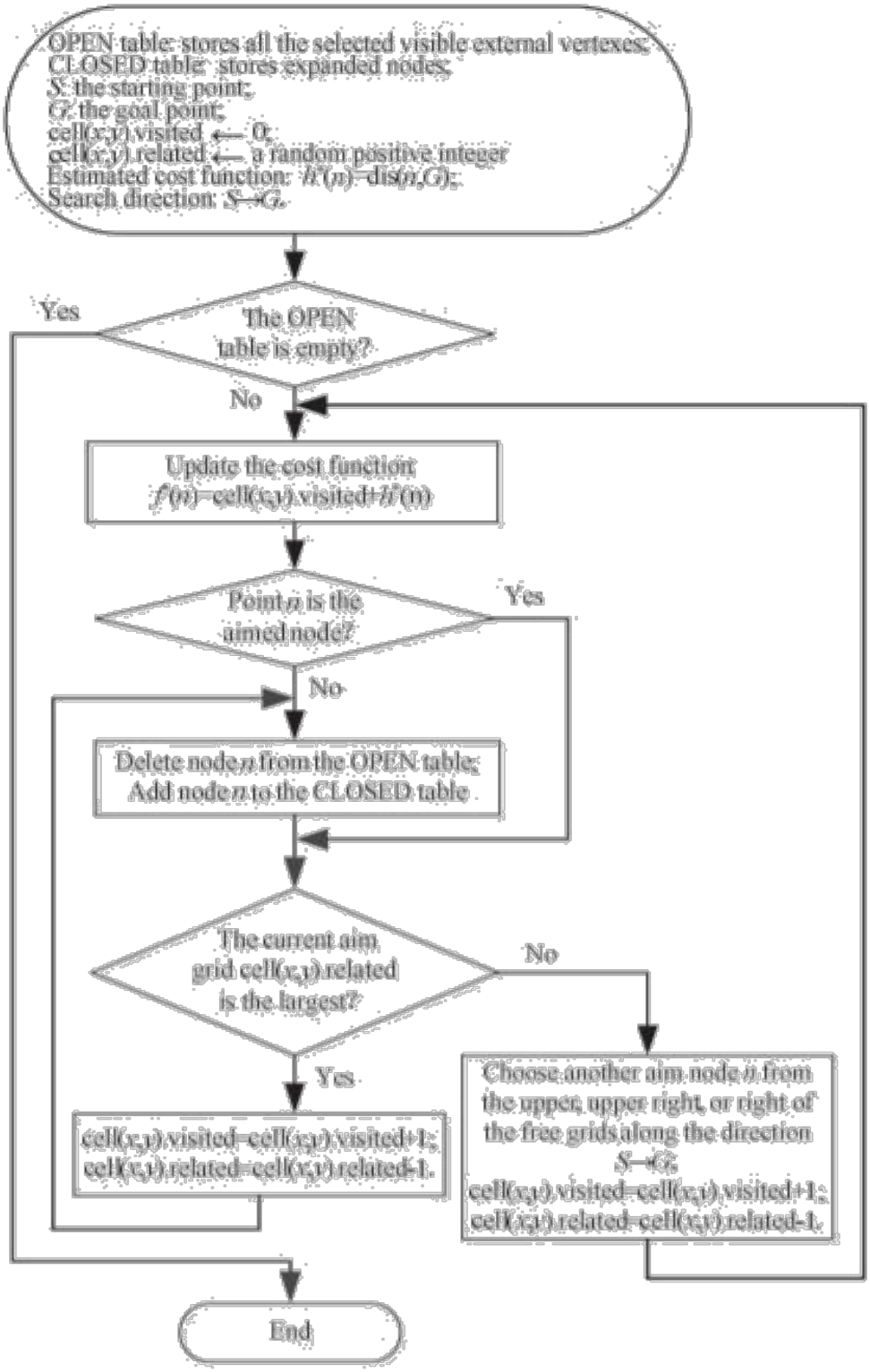


Fig 5.4: Flow chart of A* algorithm

5.1.3 Artificial Potential Field Method

A potential field algorithm uses the artificial potential field to regulate a robot around in a certain space. For our ease, we consider a space to be divided into a grid of cells with obstacles and a goal node.

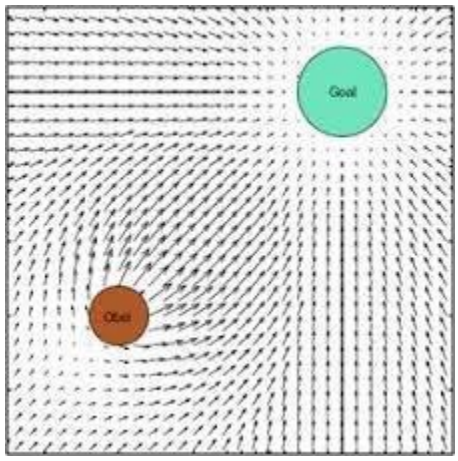


Fig 5.5: Potential Fields generated in a 2D Space

The algorithm assigns an artificial potential field to every point in the world using the potential field functions which will be described further in the explanation. The robot simulates from the highest potential to the lowest potential. Here, the goal node has the lowest potential while the starting node will have the maximum potential. Hence, we can say that the UAV moves from lowest to the highest potential.

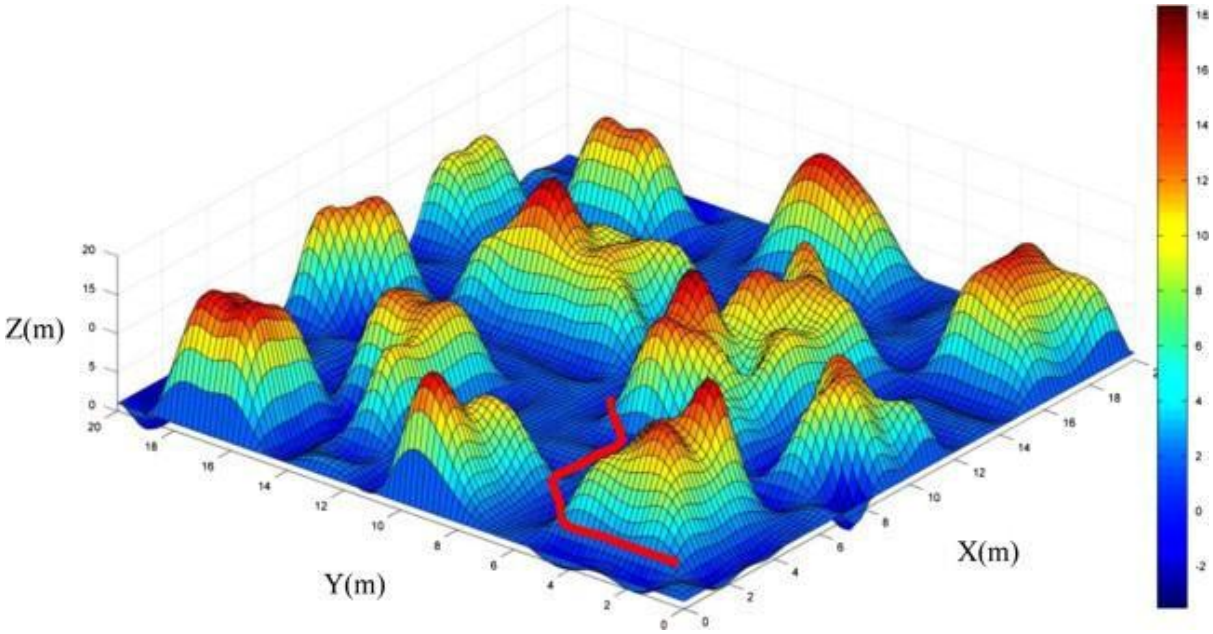


Fig 5.6: Obstacle Detection in a 3D Space

5.1.4 Visibility Graph Method

In computational geometry and robot motion planning, a visibility graph is a graph of intervisible locations, typically for a set of points and obstacles in the Euclidean plane. Each node in the graph represents a point location, and each edge represents a visible connection between them. That is, if the line segment connecting two locations does not pass through any obstacle, an edge is drawn between them in the graph. When the set of locations lies in a line, this can be understood as an ordered series. Visibility graphs have therefore been extended to the realm of time series analysis.

Visibility graphs may be used to find Euclidean shortest paths among a set of polygonal obstacles in the plane: the shortest path between two obstacles follows straight line segments except at the vertices of the obstacles, where it may turn, so the Euclidean shortest path is the shortest path in a visibility graph that has as its nodes the start and destination points and the vertices of the obstacles.

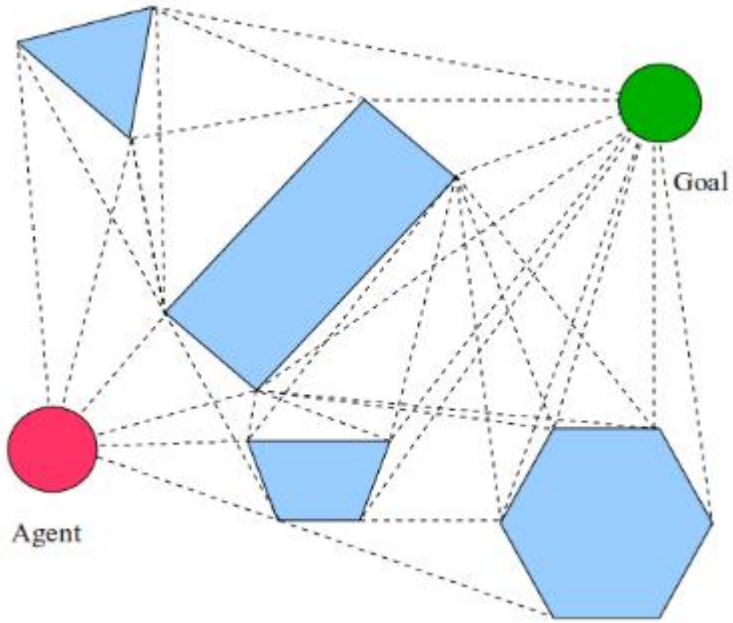


Fig 5.7: Visible lines to vertex

The shortest path between any two points that avoids a set of polygon obstacles is a polygonal curve, whose vertices are either vertices of obstacles or points ‘s’ and ‘t’.

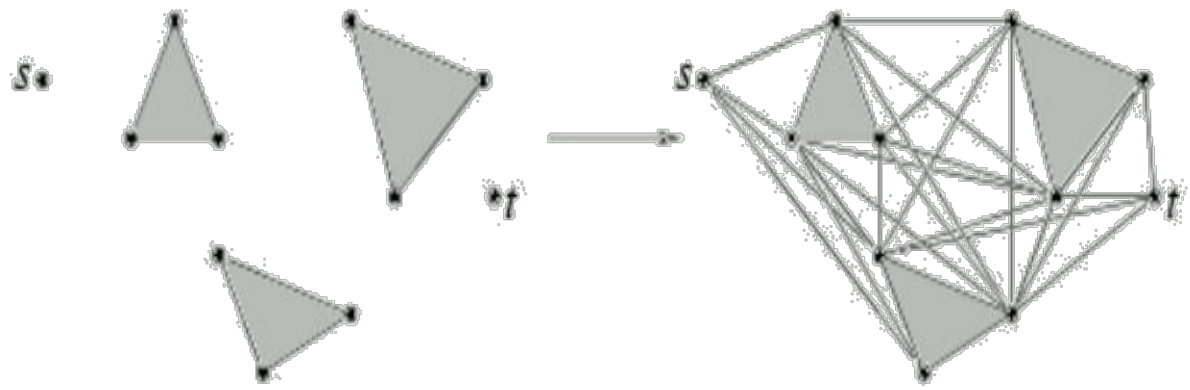


Fig 5.8: Shortest path between 2 points

5.2 Occupancy Grid Methods

Method that is using occupancy grid divides area into cells (e.g. map pixels) and assign them as occupied or free. One of cells is marked as robot position and another as a destination. Finding the trajectory is based on finding shortest line that do not cross any of occupied cells.

In past couple of decades, occupancy grid maps have become a dominant paradigm for environment modelling in mobile robots. Occupancy grid maps are spatial representations of robot environment. They represent environments surrounded by mobile robots using fine grained, metric grids of variables that reflect the occupancy of the environment. Once achieved, they enable various key functions necessary for mobile robot navigation, such as localization, path planning, collision avoidance, etc. "The occupancy grid is a multidimensional random field that maintains stochastic estimates of the occupancy state of the cells in spatial lattice". A grid-based maps represents the environment by splitting it into large number of small cells. All these cells are uniformly sized and are usually fixed during the exploration of environment. Each cell represents the probability that the corresponding region in the environment is occupied. Occupancy values range from zero to one. An occupancy value near zero p(cell = occ)= 0 corresponds to an open cell and indicates with highest probability that the corresponding region is free of any obstacles. An occupancy value near one p(cell = occ)= 1 indicates the opposite or that the region is occupied. When no previous knowledge is available the occupancy value of each grid is initialized at 0.5 indicating the presence or absence of obstacle in that particular grid cell is equally likely. Once all grids are initialized, they are updated recursively to construct a sensor-derived map of the robot's world by interpreting the incoming range readings using probabilistic sensor models. For this, the basic assumption made that the location of mobile robot is always known.

Once the robot has localized successfully, it can be supplied with destination coordinates and uses a global planner to plot a path to those coordinates. The environment is represented internally as a 2-dimensional lossless image. The environment contains five types of entities: unexplored space, known clear space, obstacles, the inflation radius, and the robot itself. The inflation radius is a buffer that radiates outward from all obstacles. The robot treats the inflation radius as an obstacle and cannot chart paths through it. The inflation radius is equal to the robot's own radius and has the effect of ensuring that the robot always generates paths with enough space to clear obstacles. It is implemented as a buffer around obstacles instead of being

applied to the robot itself for reasons of simplicity, and achieves the same effect. Within this paradigm, the global planner uses Dijkstra's algorithm to generate a path to the goal. Once a global plan has been generated, the local planner translates this path into velocity commands for the robot's motors. It does this by creating a value function around the robot, sampling and simulating trajectories within this space, scoring each simulated trajectory based on its expected outcome, sending the highest-scoring trajectory as a velocity command to the robot, and repeating until the goal has been reached. The reason that the local planner works this way is that it was written to be very general, for use by robots that may have irregular footprints

5.2.1 Occupancy Grid Path Planning in ROS

In ROS it is possible to plan a path based on occupancy grid, e.g. the one obtained from slam_gmapping. Path planner is move_base node from move_base package.

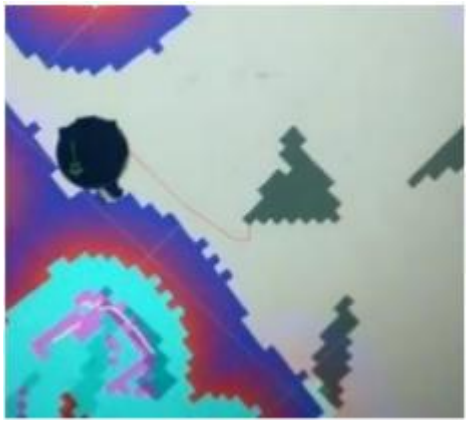


Fig. 5.9: Move base trajectry

5.2.1.1 Requirements Regarding Robot

Before continuing with move_base node certain requirements must be met, robot should:

- subscribe cmd_vel topic with message type geometry_msgs/Twist in which robot desired velocities are included.
- Publish to /tf topic transformations between robot relative to starting point and laser scanner relative to robot.
- Publish map to /map topic with message type nav_msgs/OccupancyGrid

Above configuration is met by the robot created in previous manual.

5.2.1.2 Configuration of move_base node

Move_base node creates cost map basing on occupancy grid. Cost map is a grid in which every cell gets assigned value (cost) determining distance to obstacle, where higher value means closer distance. With this map, trajectory passing cells with lowest cost is generated. Move_base node uses two cost maps, local for determining current motion and global for trajectory with longer range.
For move_base node some parameters for cost map and trajectory planner need to be defined, they are stored in .yaml files.

5.2.1.3 Common parameters for cost map

Common parameters are used both by local and global cost map. We will define following parameters:

- obstacle_range: 6.0
  In this range obstacles will be considered during path planning.
- raytrace_range: 8.5
  This parameter defines range in which area could be considered as free.
- footprint: [[0.12, 0.14], [0.12, -0.14], [-0.12, -0.14], [-0.12, 0.14]]
  This parameter defines coordinates of robot outline, this will considered during collision detecting.
- map_topic: /map
  This parameter defines topic where occupancy grid is published.
- subscribe_to_updates: true
  This parameter defines if move_base should periodically check if map was updated.
- observation_sources: laser_scan_sensor
  This parameter defines type of sensor used to provide data.
- laser_scan_sensor: {sensor_frame: laser_frame, data_type: LaserScan, topic: scan, marking: true, clearing: true}
  This parameter define properties of used sensor, these are:
  - ✓ sensor_frame - coordinate frame tied to sensor
  - ✓ data_type - type of message published by sensor
  - ✓ topic - name of topic where sensor data is published
  - ✓ marking - true if sensor can be used to mark area as occupied
  - ✓ clearing - true if sensor can be used to mark area as clear
- global_frame: map
  This parameter defines coordinate frame tied to occupancy grid map.
- robot_base_frame: base_link
  This parameter defines coordinate frame tied to robot.
- always_send_full_costmap: true
  This parameter define if costmap should be always published with complete data.

Final file should look like below:

obstacle_range: 6.0
raytrace_range: 8.5
footprint: [[0.12, 0.14], [0.12, -0.14], [-0.12, -0.14], [-0.12, 0.14]]
map_topic: /map
subscribe_to_updates: true
observation_sources: laser_scan_sensor
laser_scan_sensor: {sensor_frame: laser_frame, data_type: LaserScan, topic: scan, marking: true, clearing: true}
global_frame: map
robot_base_frame: base_link
always_send_full_costmap: true

Save it as costmap_common_params.yaml in tutorial_pkg/config directory.

5.2.1.4 Parameters for local cost map

These parameters are used only by local cost map. We will define following parameters:

- local_costmap:
  This parameter groups following parameters to be considered only by local planner.
- update_frequency: 5
  This parameter defines how often cost should be recalculated.
- publish_frequency: 5
  This parameter defines how often cost map should be published to topic.
- transform_tolerance: 0.25
  This parameter define latency in published transforms (in seconds), if transforms are older than this, planner will stop.
- static_map: false
  This parameter defines if map can change in time, true if map will not change.
- rolling_window: true
  This parameter defines if map should follow position of robot.
- width: 3
- height: 3
  These parameters define size of map (in meters).
- origin_x: -1.5
- origin_y: -1.5
  These parameters define position of left bottom map corner (in meters). If these values are half of map size, and rolling_window is set to true, then robot will always be in cost map centre.
- resolution: 0.1
  This parameter define size of single map cell (in meters).

- inflation_radius: 1.0
  This parameter defines distance to obstacle where cost should be considered, any further from obstacle than this value will be treated as no cost.

Final file should look like below:

```
local_costmap:
  update_frequency: 5
  publish_frequency: 5
  transform_tolerance: 0.25
  static_map: false
  rolling_window: true
  width: 3
  height: 3
  origin_x: -1.5
  origin_y: -1.5
  resolution: 0.1
  inflation_radius: 0.6
```

Save it as local_costmap_params.yaml in tutorial_pkg/config directory.

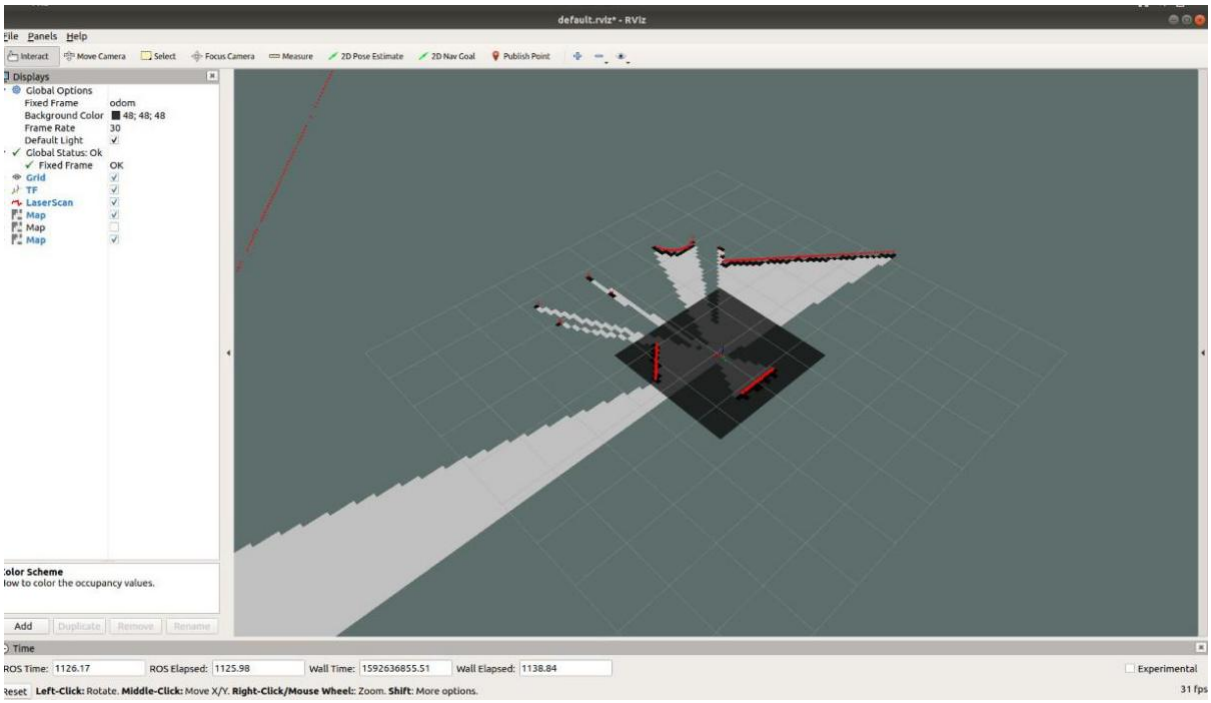


Fig. 5.10: rviz window showing local costmap

5.2.1.5 Parameters for global cost map

These parameters are used only by global cost map. Parameter meaning is the same as for local cost map, but values may be different.

Your file for global cost map should look like below:

global_costmap:
update_frequency: 2.5
publish_frequency: 2.5
transform_tolerance: 0.5
width: 15
height: 15
origin_x: -7.5
origin_y: -7.5
static_map: true
rolling_window: true
inflation_radius: 2.5
resolution: 0.1

Save it as global_costmap_params.yaml in tutorial_pkg/config directory.

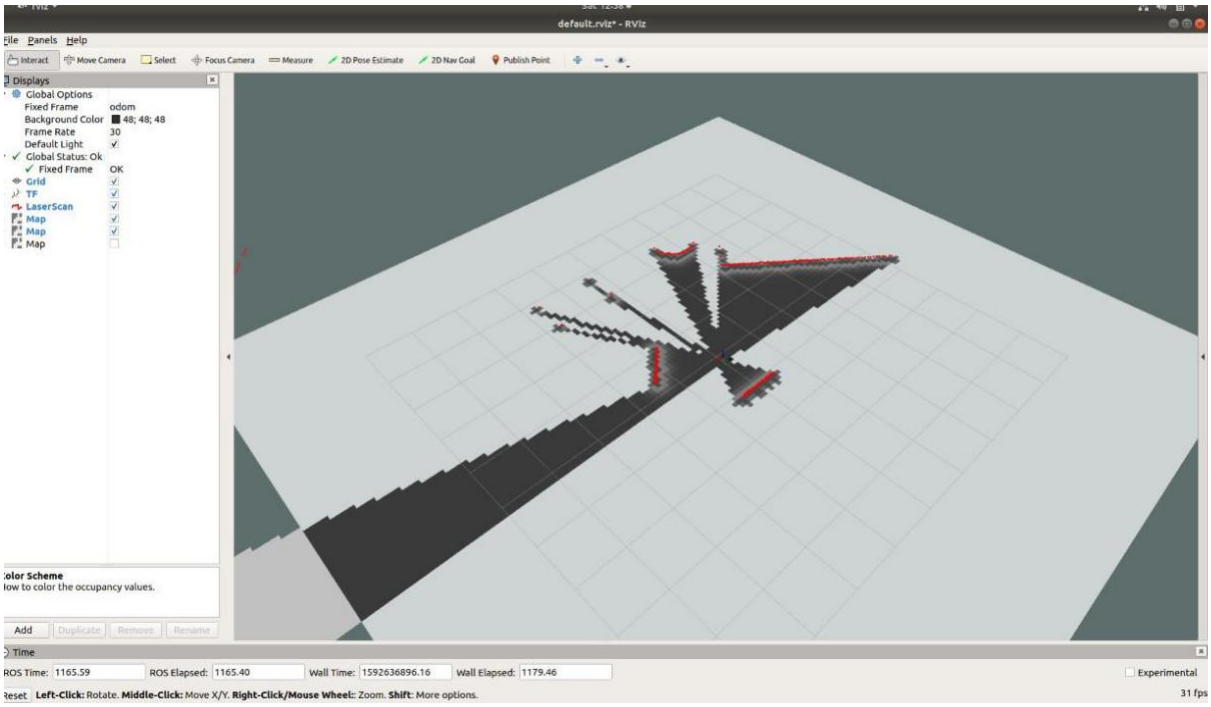


Fig. 5.11: rviz window showing global costmap

5.2.1.6 Parameters for Trajectory Planner

These parameters are used by trajectory planner. We will define following parameters:

- TrajectoryPlannerROS:
  This parameter groups following parameters to be considered only by trajectory planner.

- max_vel_x: 0.2
  This parameter defines maximum linear velocity that will be set by trajectory planner.

- min_vel_x: 0.1
  This parameter defines minimum linear velocity that will be set by trajectory planner. This should be adjusted to overcome rolling resistance and other forces that may suppress robot from moving.

- max_vel_theta: 0.35
- min_vel_theta: -0.35
  This parameter defines maximum angular velocity that will be set by trajectory planner.

- min_in_place_vel_theta: 0.25
  This parameter defines minimum angular velocity that will be set by trajectory planner. This should be adjusted to overcome rolling resistance and other forces that may suppress robot from moving.

- acc_lim_theta: 0.25
- acc_lim_x: 2.5
- acc_lim_Y: 2.5
  There parameters define maximum values of accelerations used by trajectory planner.

- holonomic_robot: false
  This parameter defines if robot is holonomic.

- meter_scoring: true
  This parameter defines if cost function arguments are expressed in map cells or meters (if true, meters are considered).

- xy_goal_tolerance: 0.15
- yaw_goal_tolerance: 0.25
  These parameters define how far from destination it can be considered as reached. Linear tolerance is in meters, angular tolerance is in radians.

Final file should look like below:

```
TrajectoryPlannerROS:
  max_vel_x: 0.2
  min_vel_x: 0.1
  max_vel_theta: 0.35
  min_vel_theta: -0.35
  min_in_place_vel_theta: 0.25

  acc_lim_theta: 0.25
```

acc_lim_x: 2.5
acc_lim_Y: 2.5

holonomic_robot: false

meter_scoring: true

xy_goal_tolerance: 0.15
yaw_goal_tolerance: 0.25

Save it as trajectory_planner.yaml in tutorial_pkg/config directory.

We may need to adjust cost map and trajectory planner parameters to your robot and area that you want to explore.

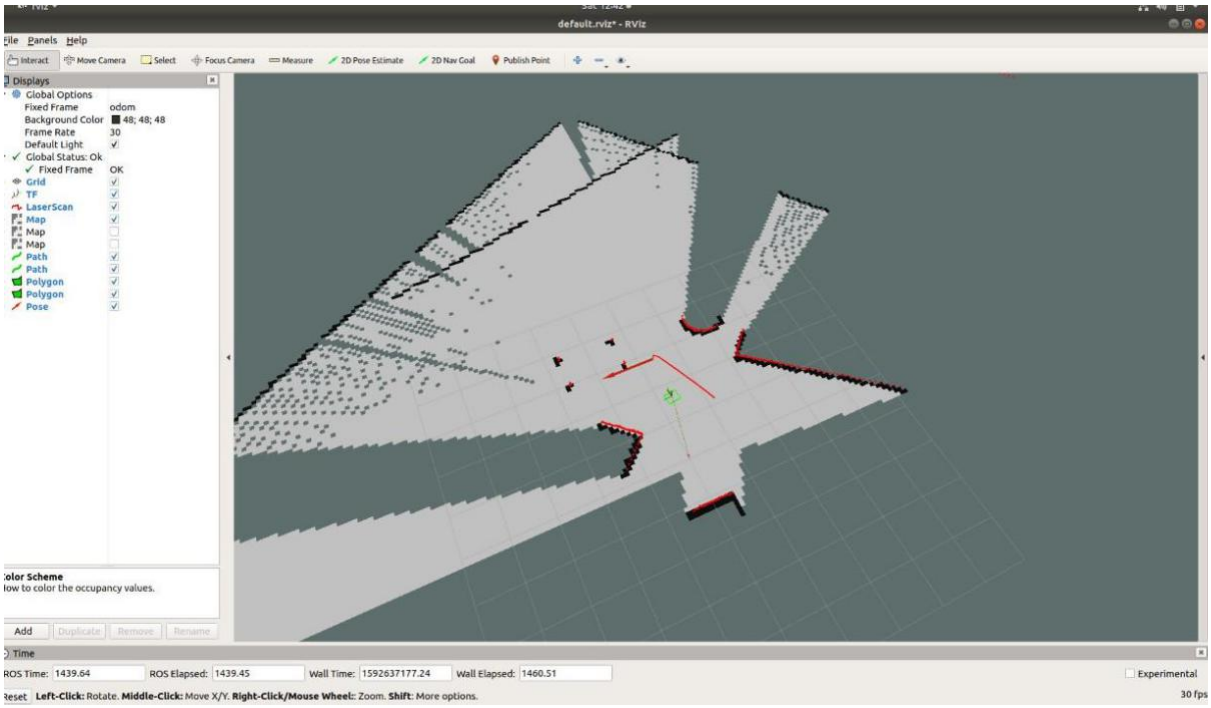


Fig. 5.12: rviz window showing path trajectory (red line path)

5.2.1.7 Launching Path Planning Node

To test above configuration, we need to run move_base node with nodes from SLAM configuration, we do not need only teleop_twist_keyboard as commands now will be issued by trajectory planner.

To sum up, we need to run following nodes:

- roslaunch rosbot_gazebo maze_world.launch
- static_transform_publisher - tf publisher for transformation of laser scanner relative to robot
- slam_gmapping - map building node
- move_base - trajectory planner
- rviz - visualization tool

For the move_base node we need to specify paths for .yaml configuration files.

- Set frequency for trajectory generation:
  <param name="controller_frequency" value="10.0"/>y
- Load common parameters for global cost map:
  <rosparam file="$(find tutorial_pkg)/config/costmap_common_params.yaml" command="load" ns="global_costmap" />
- Load common parameters for local cost map:
  <rosparam file="$(find tutorial_pkg)/config/costmap_common_params.yaml" command="load" ns="local_costmap" />
- Load only local cost map parameters:
  <rosparam file="$(find tutorial_pkg)/config/local_costmap_params.yaml" command="load" />
- Load only global cost map parameters:
  <rosparam file="$(find tutorial_pkg)/config/global_costmap_params.yaml" command="load" />
- Load trajectory planner parameters:
  <rosparam file="$(find tutorial_pkg)/config/trajectory_planner.yaml" command="load" />

We can use below launch file:

```
<launch>

  <include file="$(find rosbot_gazebo)/launch/maze_world.launch" />
  <include file="$(find rosbot_description)/launch/rosbot_gazebo.launch"/>
  <param name="use_sim_time" value="true" />

  <node pkg="tf" type="static_transform_publisher" name="laser_broadcaster" args="0
0 0 3.14 0 0 base_link laser 100" />
  <node pkg="tf" type="static_transform_publisher" name="camera_publisher" args="-
0.03 0 0.18 0 0 0 base_link camera_link 100" />

  <node pkg="gmapping" type="slam_gmapping" name="gmapping_node" output="log">
    <param name="base_frame" value="base_link" />
    <param name="odom_frame" value="odom" />
    <param name="delta" value="0.01" /> <param
    name="xmin" value="-5" /> <param
    name="ymin" value="-5" /> <param
    name="xmax" value="5" /> <param
    name="ymax" value="5" />
    <param name="maxUrange" value="5" />
    <param name="map_update_interval" value="1" />
    <param name="linearUpdate" value="0.05" />
    <param name="angularUpdate" value="0.05" />
```

```
    <param name="temporalUpdate" value="0.1" />
    <param name="particles" value="100" />
  </node>

  <node pkg="move_base" type="move_base" name="move_base" output="log">
    <param name="controller_frequency" value="10.0" />
   <rosparam file="$(find tutorial_pkg)/config/costmap_common_params.yaml"
    command="load" ns="global_costmap" />
    <rosparam file="$(find tutorial_pkg)/config/costmap_common_params.yaml"
    command="load" ns="local_costmap" />
    <rosparam file="$(find
    tutorial_pkg)/config/local_costmap_params.yaml" command="load" />
    <rosparam file="$(find tutorial_pkg)/config/global_costmap_params.yaml"
    command="load" />
    <rosparam file="$(find tutorial_pkg)/config/trajectory_planner.yaml"
    command="load" />
  </node>

  <node pkg="rviz" type="rviz" name="rviz" args="-d $(find
   tutorial_pkg)/rviz/tutorial_7.rviz"/>


</launch>
```

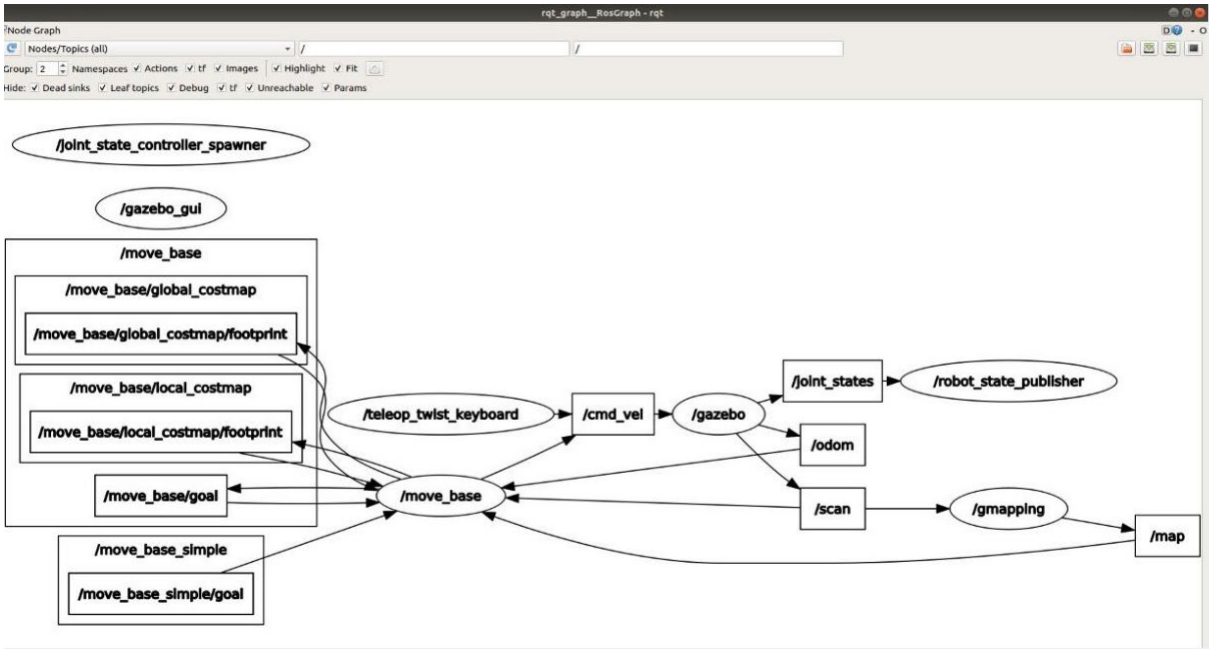


Fig. 5.13: rqt_graph

5.2.1.8 Setting Goal for Trajectory Planner

After we launched trajectory planner with all accompanying nodes, our robot still will not be moving anywhere. We have to specify target position first.

We will begin with visualization of robot surrounding, cost maps and planned trajectory. Go to rviz, add Tf, /scan and /map, again open object adding window, go to tab “By topic” and from the list select

- move_base/TrajectoryPlannerROS/local_plan/Path
- move_base/TrajectoryPlannerROS/global_plan/Path
- move_base/global_costmap/footprint/Polygon

Then for global plan path change its colour to red (values 255; 0; 0):

Now we can add one more element, open object adding window, go to tab “By topic” and from the list select /move_base_simple/goal/Pose, this will visualize destination point for your robot, it will not appear until you set destination.

If you want to observe obstacles that are considered in path planning you may add two more objects, these will be local and global costmaps, open object adding window, go to tab “By topic” and from the list select move_base/global_costmap/costmap and move_base /local_costmap/costmap. Now change parameter Color Scheme of both costmaps to costmap, this will allow to distinguish costmaps from occupancy grid map.

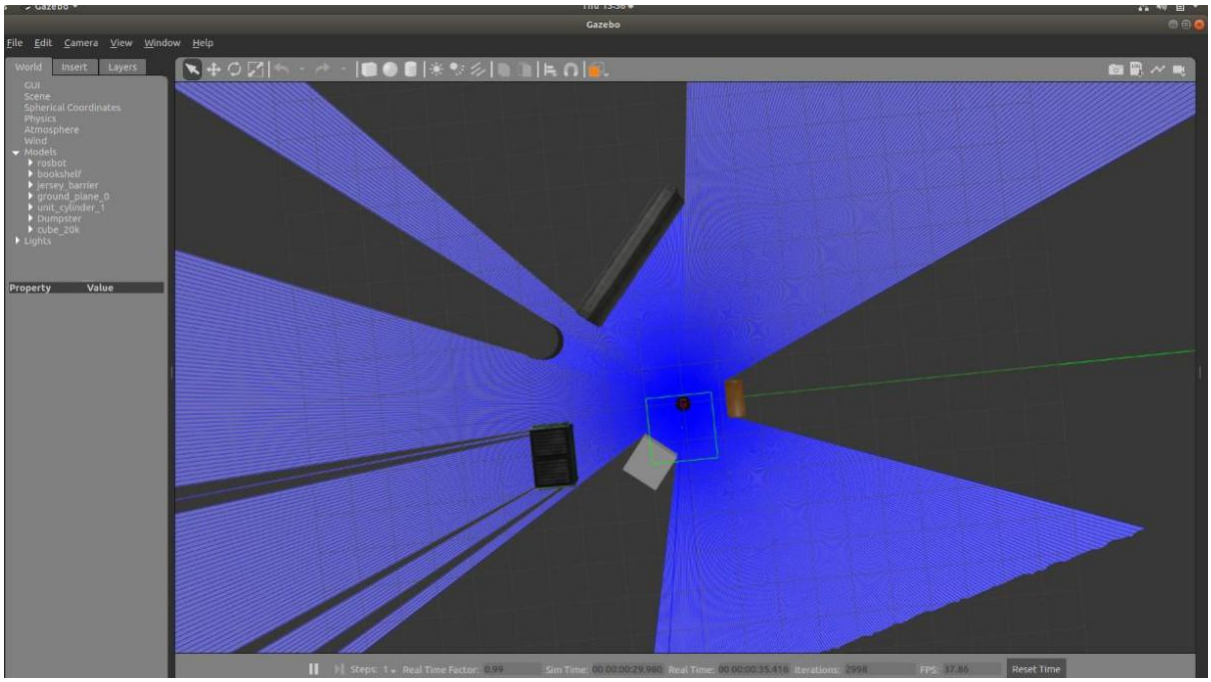


Fig. 5.14: Gazebo Window

Having all the elements visualized, you can set goal for robot, from Toolbar click button 2D nav goal, then click a place in Visualization window, that will be destination for your robot. Observe as path is generated (you should see a line from your robot pointing to direction) and robot is moving to its destination.

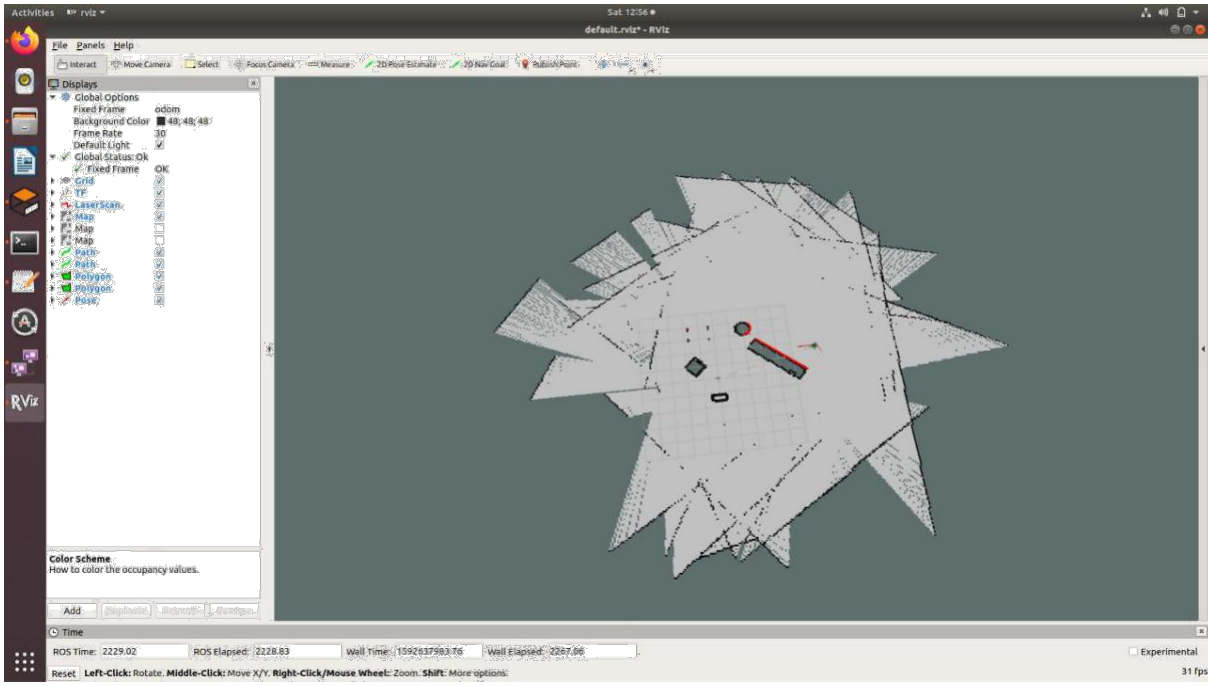


Fig. 5.15: Generated map after manually setting destination points

Chapter-6

UNKNOWN ENVIRONMENT EXPLORATION

6.1 Introduction

While many robots can navigate using maps, few can build their own maps. Usually a human must map the territory in advance, providing either the exact locations of obstacles (for metric maps) or a graph representing the connectivity between open regions (for topological maps). As a result, most mobile robots become unable to navigate efficiently when placed in unknown environments.

Exploration has the potential to free robots from this limitation. We define exploration to be the act of moving through an unknown environment while building a map that can be used for subsequent navigation. A good exploration strategy is one that generates a complete or nearly complete map in a reasonable amount of time.

Considerable work has been done in simulated exploration but these simulations often view the world as a set of floorplans. A blueprint view of a typical office building represent a structure that seem simple and straightforward- rectangular offices, square conference rooms, straight hallways, and right angles everywhere- but the reality is often quite different. A real mobile robot may have to navigate through rooms cluttered with furniture, where walls may be hidden behind desks and bookshelves.

A few researchers have implemented exploration system using real robots. These robots have performed well but only within environments that satisfy certain restrictive assumptions. For example some systems are limited to environments that can be explored using wall-following while others require that all walls intersect at right angles and that these walls be unobstructed and visible to the robot. Some indoor environments meet this requirement but many do not.

Our goal is to develop exploration strategies for the complex environments typically found in real office buildings. Our approach is based on the detection of frontiers, regions on the border between space known to be open and unexplored space. In this project, we describe how to detect frontiers in occupancy grids and how to use frontiers to guide exploration.

6.2 Frontier-Based Exploration

The central question in exploration is: Given what you know about the world, where should you move to gain as much new information as possible? Initially, you know nothing except what you can see from where you are standing. You want to build a map that describes as much of the world as possible, and you want to build this map as quickly as possible.

The central idea behind frontier-based exploration is: to gain the most new information about the world, to move the boundary between open space and uncharted territories.

Frontiers are regions on the boundary between open space and unexplored space. When a robot moves to a frontier, it can see into unexplored space and add a new information to its map. As a result, the mapped territory expands, pushing back the boundaries between the known and unknown. By moving to the successive frontier, the robot can constantly increase its knowledge of the world. We call this strategy Frontier based exploration.

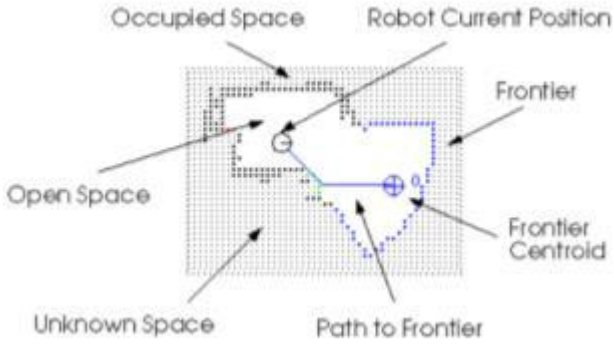


Fig. 6.1: Frontiers

If a robot with a perfect map could navigate to a particular point in space, that point is considered accessible. All accessible space is contigeous, since a path must exist from the robot's initial position to any accessible point. Every such path will be at least partially in unknown territory will cross a frontier. When a robot navigates to that frontier, it will incorporate more of the space covered by a path into mapped territory. I the robot does not incorporate the entire path at one time, then a new frontier will always exist further along the path separating the known and unknown segments and providing a new destination for exploration.

In this way, a robot using frontier-based exploration eventually explore all of the accessible space in the world- assume perfect sensors and the perfect motor control. A zeno-like paradox where the new information contributed by each new frontier decreases geometrically is theoretically possible (though highly unlikely), but even in such a case the map will become arbitrary accurate in a finite amount of time.

A real question is how well a frontier-based exploration will work using the noisy sensor and imperfect motor control of a real robot in the real world.

6.3 Frontier Detection

After an evidence grid has been constructed, each cell in the grid is classified by comparing its occupancy probability to the initial(prior) probability assigned to all cells. This algorithm is not particularly sensitive to a specific value of this prior probability (0.5).

Each cell is placed into one of three classes:

- **Open**: occupancy probability <prior probability
- **Unknown**: occupancy probability =prior probability
- **Occupied**: occupancy probability >prior probability

A process analogous to edge detection and region extraction in computer vision is used to find the boundaries between the open space and unknown space. Any open cell adjacent to an unknown cell is labelled a frontier edge cell. Adjacent edge cells are grouped into frontier regions. Any frontier region above a certain minimum size (roughly the size of the robot) is considered a frontier.

6.4 Navigating to Frontiers

Once frontiers have been detected within a particular evidence grid, the robot attempts to navigate the nearest accessible, unvisited frontier. The path planner uses a depth-first search on the grid, starting at the robot's current cell and attempting to take the shortest obstacle-free path to the cell containing the goal location.

While the robot moves toward its destination, reactive obstacles avoidance behaviours prevents collision with any obstacles not present while the evidence grid was constructed. In a dynamic environment this is necessary to avoid collision with, for example, people who are walking about. These behaviours allow the robot to steer around these obstacles and as long as the world has not change too drastically, return to follow its path to the destination.

When the robot reaches its destination. That location is added to the list of previously visited frontiers. The robot performs a 360 sensor sweep using laser-limited sonar and adds a new information to the evidence grid. Then the robot detects frontiers present in the updated grid and attempts to navigate to the nearest accessible, unvisited frontier.

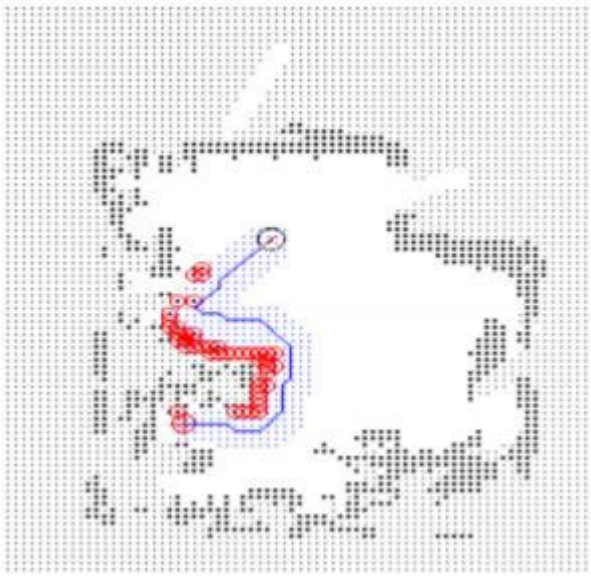


Fig. 6.2: Path planning to next Frontier

If the robot is unable to make progress toward its destination, then after a certain amount of time, the robot will determine that the destination is inaccessible, and its location will be added to the list of inaccessible frontiers. The robot will then conduct a sensor sweep, update the evidence grid, and attempt to navigate to a closest remaining accessible, unvisited frontiers.

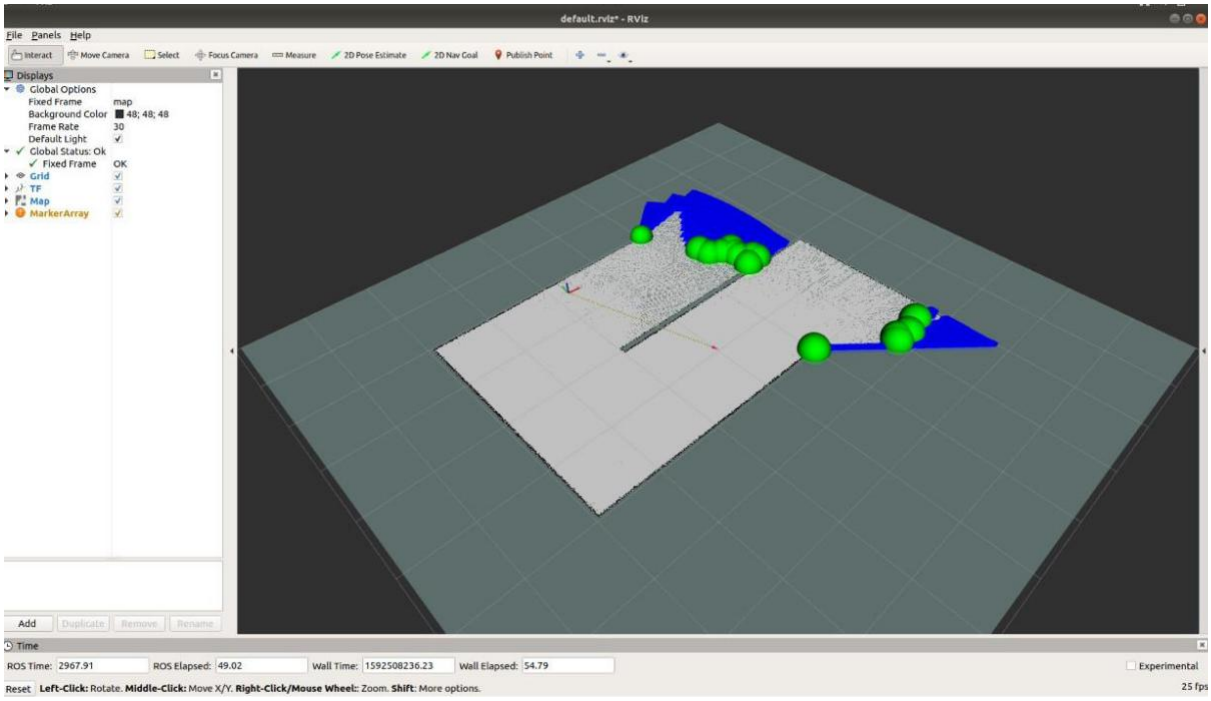


Fig. 6.3: rviz Window showing Frontiers

6.5 Exploration using ROS

Environment exploration task is to determine robot desired positions in such a sequence which gives as much information regarding environment as possible. One of the approaches for this task is to frontiers of occupancy grid. As a frontier we define line between free space ant territory marked as unknown. Moving towards frontiers, unknown area can be explored and marked as free or occupied and frontiers are moved into an unknown territory. Process is repeated until all frontiers are investigated, this means free area must be surrounded by occupied cells.

In ROS it is possible to explore environment with use of occupancy grid frontiers. One of the nodes, that perform this task is explore node from explore_lite package. This node uses occupancy grid e.g. created by slam_gmapping and publishes goal position to /move_base/goal topic subscribed by path planner e.g. move_base node.

6.5.1 Requirements regarding Robot

Before continuing with explore_lite node certain requirements must be met, robot should:

- subscribe /move_base/goal topic with message type geometry_msgs/PoseStamped in which robot desired positions are included.
- Publish map to /map topic with message type nav_msgs/OccupancyGrid

6.5.2 Configuration of Explore Node

Before starting experimenting with explore_lite, we need to have working move_base for navigation. We should be able to navigate with move_base manually through rviz.
We should be also able to to navigate with move_base though unknown space in the map. If we set the goal to unknown place in the map, planning and navigating should work. With most planners this should work by default. Navigation through unknown space is required for explore_lite.

If we want to use costmap provided by move_base we need to enable unknown space tracking by setting track_unknown_space: true.

All required configurations are set properly in the previous chapter, so we can start experimenting with explore_lite. Provided explore.launch should work out-of-the box in most cases, but as always we might need to adjust topic names and frame names according to our setup.

Based on occupancy grid, explore node determines frontiers between free and unknown area and using them determines robot destinations.

6.5.3 Parameters

- robot_base_frame - The name of the base frame of the robot. This is used for determining robot position on map.
- costmap_topic - Specifies topic of source nav_msgs/OccupancyGrid.
- costmap_updates_topic - Specifies topic of source map_msgs/OccupancyGridUpdate. Not necessary if source of map is always publishing full updates, i.e. does not provide this topic.
- visualize - Specifies whether or not publish visualized frontiers.
- planner_frequency - Rate in Hz at which new frontiers will computed and goal reconsidered.
- progress_timeout - Time in seconds. When robot do not make any progress for progress_timeout, current goal will be abandoned.
- potential_scale - Used for weighting frontiers. This multiplicative parameter affects frontier potential component of the frontier weight (distance to frontier).
- orientation_scale - Used for weighting frontiers. This multiplicative parameter affects frontier orientation component of the frontier weight.
- gain_scale - Used for weighting frontiers. This multiplicative parameter affects frontier gain component of the frontier weight (frontier size).
- transform_tolerance - Transform tolerance to use when transforming robot pose.
- min_frontier_size - Minimum size of the frontier to consider the frontier as the exploration goal. Value is in meter.

Save configuration as exploration.yaml in tutorial_pkg/config directory.

Final file look like below:

robot_base_frame: base_link
costmap_topic: map
costmap_updates_topic: map_updates
visualize: true
planner_frequency: 0.33
progress_timeout: 30.0
potential_scale: 3.0
orientation_scale: 0.0
gain_scale: 1.0
transform_tolerance: 0.3
min_frontier_size: 0.75

6.5.4 Launching Exploration Task

To test above configuration, we need to run explore_lite node with nodes from path planning configuration.

We need to run following nodes:

- roslaunch rosbot_gazebo maze_world.launch
- static_transform_publisher - tf publisher for transformation of laser scanner relative to robot
- slam_gmapping - map building node
- move_base - trajectory planner
- explore_lite - exploration task
- rviz - visualization tool

For the explore_lite node you will need to specify path for .yaml configuration file:

```
<node pkg="explore_lite" type="explore" respawn="true" name="explore"
   output="screen">
      <rosparam file="$(find tutorial_pkg)/config/exploration.yaml" command="load"
/> </node>
```

We can use below launch file and save it as tutorial_8.launch:

```
<launch>

   <include file="$(find rosbot_gazebo)/launch/maze_world.launch" />
   <include file="$(find rosbot_description)/launch/rosbot_gazebo.launch"/>
   <param name="use_sim_time" value="true" />

   <node pkg="tf" type="static_transform_publisher" name="laser_broadcaster" args="0
   0 0 3.14 0 0 base_link laser 100" />
   <node pkg="tf" type="static_transform_publisher" name="camera_publisher" args="-
   0.03 0 0.18 0 0 0 base_link camera_link 100" />

   <node pkg="gmapping" type="slam_gmapping" name="gmapping_node" output="log">
```

```
    <param name="base_frame" value="base_link"
    /> <param name="odom_frame" value="odom" />
    <param name="delta" value="0.01" /> <param
    name="xmin" value="-5" />
    <param name="ymin" value="-5" />
    <param name="xmax" value="5" />
    <param name="ymax" value="5" />
    <param name="maxUrange" value="5" />
    <param name="map_update_interval" value="1" />
    <param name="linearUpdate" value="0.05" />
    <param name="angularUpdate" value="0.05" />
    <param name="temporalUpdate" value="0.1" />
    <param name="particles" value="100" />
  </node>

  <node pkg="move_base" type="move_base" name="move_base" output="log">
    <param name="controller_frequency" value="10.0" />
   <rosparam file="$(find tutorial_pkg)/config/costmap_common_params.yaml"
    command="load" ns="global_costmap" />
    <rosparam file="$(find tutorial_pkg)/config/costmap_common_params.yaml"
    command="load" ns="local_costmap" />
    <rosparam file="$(find
    tutorial_pkg)/config/local_costmap_params.yaml" command="load" />
    <rosparam file="$(find tutorial_pkg)/config/global_costmap_params.yaml"
    command="load" />
    <rosparam file="$(find tutorial_pkg)/config/trajectory_planner.yaml"
    command="load" />
  </node>

  <node pkg="explore_lite" type="explore" respawn="true" name="explore"
  output="screen">
    <rosparam file="$(find tutorial_pkg)/config/exploration.yaml" command="load"
  /> </node>

  <node pkg="rviz" type="rviz" name="rviz" args="-d $(find
  tutorial_pkg)/rviz/tutorial_8.rviz"/>


</launch>
```

Launch this with command:

roslaunch tutorial_pkg tutorial_8.launch

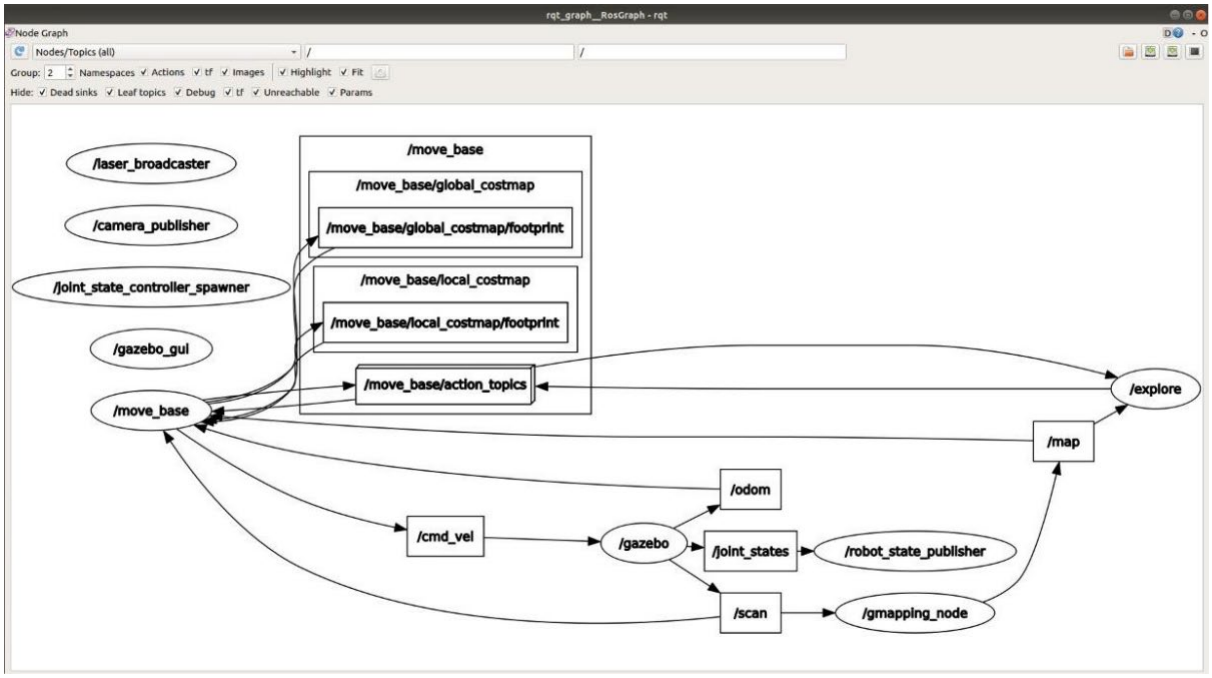


Fig. 6.4: rqt_graph

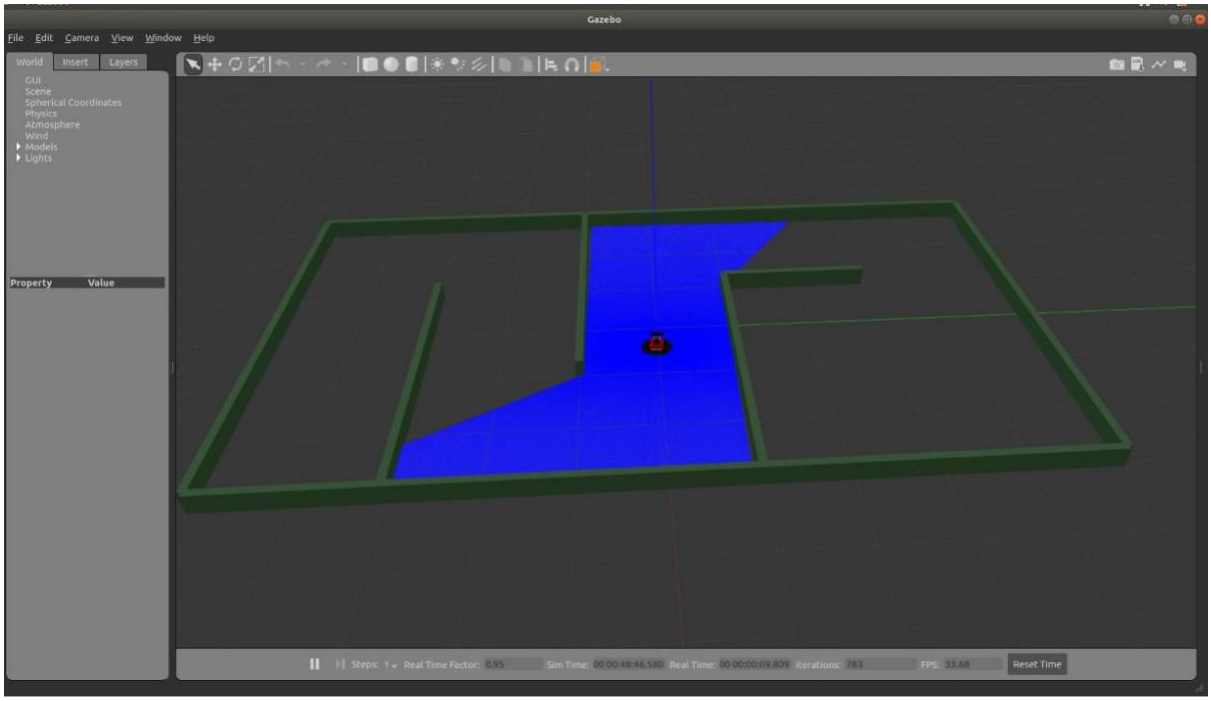


Fig. 6.5: Gazebo Window showing Maze

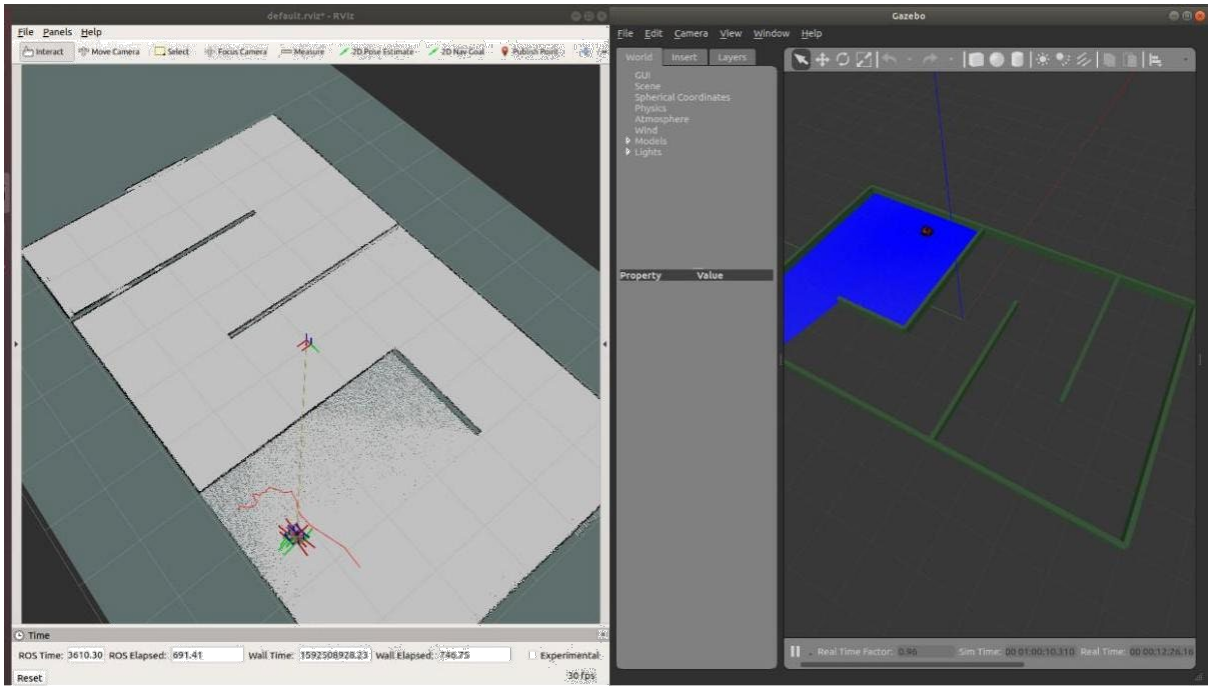


Fig. 6.6: Exploration

6.5.5 Explore

If everything was set correctly exploration will start immediately after node initialization. Exploration will finish when whole area is discovered.

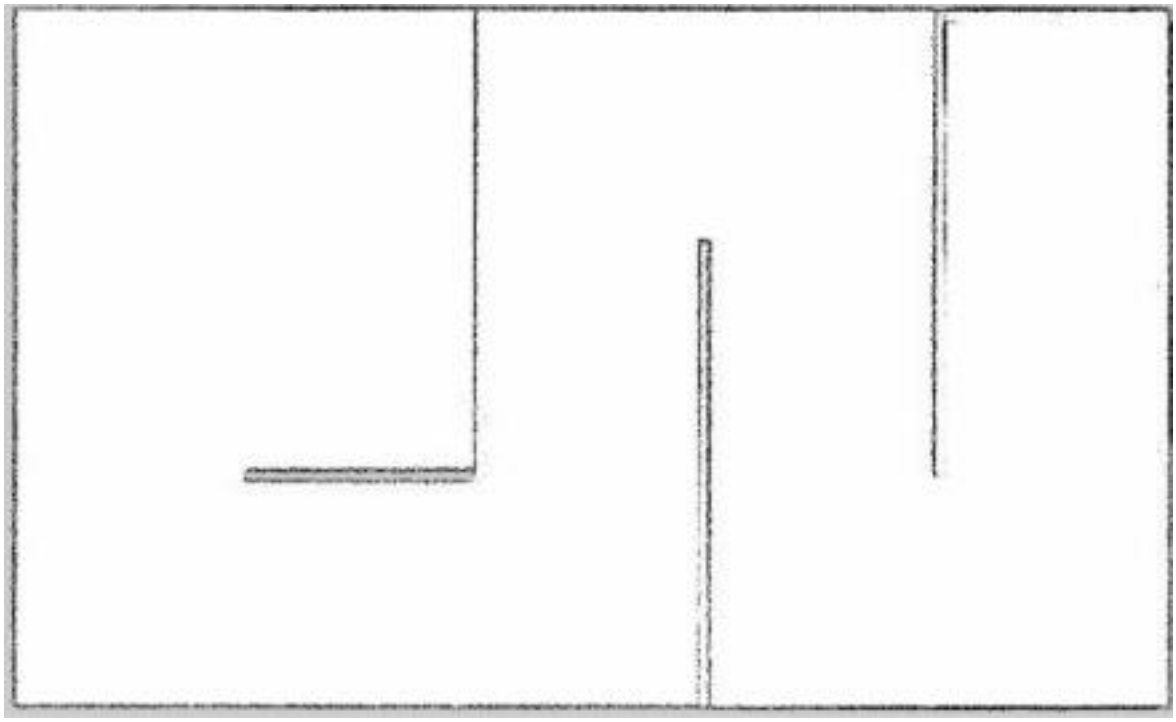


Fig 6.7: Generated Map after Exploration

Chapter-7

APPLICATION AREAS

Autonomous robots are used to explore environments that are difficult for human beings to explore. Some of the application areas are discussed below:

1. Space: Missions to planets and other places in space, or investigations of dangerous sites like radioactive environments.
2. Underwater: Autonomous underwater vehicles to explore sea surface and study about marine creatures
3. Autonomous robots can be used as guides in buildings to show directions
4. At Home: Intelligent wheelchairs, vacuum cleaners
5. Underground: Exploration of mines
6. Patrolling: SLAM bots are used for security purposes in army and other confidential areas.
7. Sanitization in Hospitals

Chapter-8

SUMMARY

The Simultaneous Localization and Mapping (SLAM) problem is concerned with building an intelligent robot that can identify its position when kept at an unknown location and in an unknown environment, while incrementally building a map of the environment it is placed in. SLAM has been formulated and solved by the Robotics community and several algorithms (for example, self-driving cars) already exist. However, the cost of sensors involved and computational intensity of the algorithms are pretty high.

The aim of this project is to build a self-aware, mobile 3D-motion and depth sensing robot using the Raspberry Pi Hardware platform. With the advent of cloud computing, the barriers of high-performance computing have been greatly reduced. Utilizing this ability to offload signal processing, the project intends to produce a single robot implementation with minimal cost on the bill of materials and computational cost. This requires using an architecture which can effectively decouple the robot from the server with minimal effect on the robot's processing capability.

We have demonstrated an affordable implementation platform using low cost computational hardware and open-source algorithms. Utilizing the wrappers offered by Robot Operating System [ROS] a server-node based model, utilizing a Linux-based laptop as a server and a Raspberry-Pi based robot as the client, has been successfully built. A real-time monocular Visual SLAM system on the above described architecture has been successfully demonstrated and results compared.

Chapter-9

FUTURE SCOPE

Besides discussing many accomplishments and future challenges for the SLAM community, we also examined opportunities connected to the use of novel sensors, new tools (e.g., convex relaxations and duality theory, or deep learning), and the role of active sensing.

SLAM still constitutes an indispensable backbone for most robotics applications and despite the amazing progress over the past decades, existing SLAM systems are far from providing insightful, actionable, and compact models of the environment, comparable to the ones effortlessly created and used by humans.

Coming to the future, I see SLAM still being relevant with added improvements in methods as more research is done in the field. A little farther ahead in the future there might be techniques that take a new perspective on the problem and solve it better, but I think that will be really far in the future (say 20–40 years) and for that to be implemented in mainstream products would take even longer. SLAM would still be relevant but in a different form than we are familiar with today. Since SLAM is such a fundamental problem it would still be relevant similar to how understanding Newton's gravitational model is still relevant to understanding relativistic models.

So, in the future we see it being used extensively for a lot of autonomous navigation tasks even more than it is now.

Chapter-10

REFERENCES

1. Yamauchi, B. A frontier-based approach for autonomous exploration. In Proceedings of the 1997 IEEE International Symposium on Computational Intelligence in Robotics and Automation, Monterey, CA, USA, 10–11 July 1997; pp. 146–151.
2. Lozano-Perez, T. Automatic planning of manipulator transfer movements. IEEE Trans. Syst. Man Cybern. 1981, 11, 681–698. [CrossRef]
3. Moravec, H.P.; Elfes, A. High resolution maps from wide angle sonar. In Proceedings of the IEEE Conference on Robotics and Automation, St. Louis, MO, USA, 25–28 March 1985; pp. 19–24.
4. Iyengar, S.S.; Jorgensen, C.; Rao, S.; Weisbin, C.R. Learned Navigation Paths for a Robot in Unexplored Terrain; Technical Report; Oak Ridge National Lab.: Oak Ridge, TN, USA, 1985.
5. Amigoni, F.; Gallo, A. A multi-objective exploration strategy for mobile robots. In Proceedings of the 2005 IEEE International Conference on Robotics and Automation, Barcelona, Spain, 18–22 April 2005; pp. 3850–3855
6. Chatila, R.; Laumond, J.P. Position referencing and consistent world modeling for mobile robots. In Proceedings of the 1985 IEEE International Conference on Robotics and Automation, St. Louis, MO, USA, 25–28 March 1985; Volume 2, pp. 138–145.
7. Dudek, G.; Jenkin, M.; Milios, E.; Wilkes, D. Robotic exploration as graph construction. IEEE Trans. Robot. Autom. 1991, 7, 859–865. [CrossRef]
8. Kuipers, B.; Byun, Y.T. A robot exploration and mapping strategy based on a semantic hierarchy of spatial representations. Robot. Autonom. Syst. 1991, 8, 47–63. [CrossRef]
9. Edlinger, T.; von Puttkamer, E. Exploration of an indoor-environment by an autonomous mobile robot. In Proceedings of the IEEE/RSJ/GI International Conference on Intelligent Robots and Systems' 94.'Advanced Robotic Systems and the Real World', IROS'94, Munich, Germany, 12–16 September 1994; Volume 2, pp. 1278–1284.
10. Makarenko, A.; Williams, S.B.; Bourgault, F.; Durrant-Whyte, H.F. An experiment in integrated exploration. In Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS, Lausanne, Switzerland, 30 September–4 October 2002; pp. 534–539.