Fiki Jusuf

# Auto-Navigation For Robots. Implementation of ROS

Helsinki Metropolia University of Applied Sciences

Bachelor of Engineering

Automation Engineering

Bachelor’s Thesis

16 March 2016

| | |
|---|---|
| Author(s)<br>Title<br><br>Number of Pages<br>Date | Fiki Jusuf<br>Auto-Navigation For Robots. Implementation of ROS In Simulation<br>25 pages + 2 appendices<br>16 March 2016 |
| Degree | Bachelor of Engineering |
| Degree Programme | Automation Engineering |
| Instructor(s) | Jari Savolainen, Senior Lecturer. |

The purpose of this thesis was to investigate how to perform and operate auto- navigation for robots. The best candidate to perform the navigation was based on Robot Operating System, ROS, developed by Willow Garage in 2007. The system was developed under BSD license and gradually has attracted more experts to implement the system. As a result, it has become a widely-used platform among researchers of robotics community. In 2013 the maintenance and the core development of ROS was transferred to Open Source Robotics Foundation and is still running until this day. Currently ROS is implemented by tens of thousands of users all around the world varying from hobby projects to large scale industrial automation systems.

A small introduction of ROS history and its operating method is discussed in the first chapter in order to understand the principle of ROS. After the introduction, a more detailed analysis of navigation in ROS is presented.

The auto-navigation performance was tested with a simulation where a TurtleBot, an open source personal research robot, is used in a simulated world. The robot was positioned in a maze map where, first, its approximate location in the map was established before giving a command to navigate autonomously to the desired end point. The robot observed with laser scanners possible obstructions and walls in order to avoid collisions and reach its destination.

Also testing Turtlebot's navigation in an unknown environment was performed. First step was to create a map using SLAM method by controlling TurtleBot manually. After the map was created, TurtleBot managed to perform auto-navigation on the created map.

In this thesis, also the installation of ROS and the simulation package in Linux Ubuntu OS is explained step-by-step before describing the performed simulation test and presenting the results.

Finally, a conclusion of auto-navigation performance is discussed and analyzed based on the results.

| | |
|---|---|
| Keywords | ROS, Navigation, Simulation, Robotics, TurtleBot |

| | |
|---|---|
| Tekijä(t)<br>Otsikko<br><br>Sivumäärä<br>Päivämäärä | Fiki Jusuf<br>Autonavigointi roboteille. ROS sovelluksen käyttö simuloinnissa.<br><br>25 sivua + 2 liitettä<br>16. maaliskuuta 2016 |
| Tutkinto | Insinööri (AMK) |
| Koulutusohjelma | Automaatiotekniikka |
| Ohjaaja(t) | Jari Savolainen, Lehtori |

Tämän lopputyön tarkoitus on tutustua ja perehtyä robottien autonavigointiin. Paras vaihtoehto autonavigointisovellukseen perustuu Robot Operating System-järjestelmään, ROS, joka on Willow Garagen kehittämä vuonna 2007. Järjestelmä kehitettiin BSD-lisenssin pohjalta ja vähitellen on houkutellut jatkuvasti lisää käyttäjiä. Tästä johtuen järjestelmä on käyttöönotettu laajasti robotiikan tutkijoiden keskuudessa. Vuonna 2013 ROSin ylläpito ja kehitys siirtyi Open Source Robotics Foundation-järjestöön ja on jatkunut tähän päivään asti.  Tällä hetkellä ROSia sovelletaan kymmenentuhannen käyttäjän piirissä ympäri maailmaa vaihdellen harrastajista ison luokan tuotantoautomaatio järjestelmän välillä.

Lyhyt johdanto ROSin historiaan ja sen operointitapaan esitellään ensimmäisessä kappaleessa, jotta ymmärretään ROSin periaatteita. Johdannon jälkeen käydään läpi yksityiskohtaisemmin ROSin navigointitoimintaan.
Autonavigoinnin suoritusta testattiin simuloinnin kautta missä TurtleBot-robotti, avoimen lähden tutkimusrobotti, käytetiin simulaatiomaailmassa. Robottia asetettiin sokkelokarttaan missä sille ensin määritettiin sijaintia, ennen kuin annettiin käsky liikkua automaattisesti haluttuun kohteeseen. Robotti tarkkaili laseritutkaimen avulla mahdollisia esteitä ja ja seinä välttääkseen yhteentörmäyksiä ja saavuttaakseen määränpäähän.
Myös TurtleBot-robotin navigoinnin testausta tuntemattomassa ympäristössä testattiin. Ensimmäinen vaihe oli kartan luominen SLAM-menetelmän avulla ohjaamalla TurtleBot-robottia manuaalisesti. Kun kartta oli luotu, TurtleBot-robotti kykeni liikkumaan autonavigoinnin avulla kehitetyssä kartassa.
ROS- ja simulaatiopaketin asennusta esitellään myös työssä ohjeiden avulla ennen kuin suoritetaan simulaatiotestausta ja esitellään tuloksia.
Lopuksi autonavigaation suorituksen tuloksista käydään läpi ja analysoidaan.

| | |
|---|---|
| Avainsanat | ROS, Navigointi, Simulaatio, Robotiikka, TurtleBot |

## Contents

## List of Abbreviations

ROS – Robot Operating System. An open source based framework for programming robot software.

BSD – Berkley Software Distribution. An open software license which gives the user the freedom to use the code and alter the program for own purpose as long as the license text is always included in the code.

ORCA – open-source based framework system for creating component-based robotics.

OpenRDK – An open-source modular software framework for developing distributed robotics systems on rapid environments.

AMCL – Adaptive Monte Carlo Localization. An algorithm based probabilistic localization system for a robot movement in a 2D environment.

SLAM – Simultaneous Localization and Mapping. A laser based scan for creating a 2-D map.

LTS – Long Term Support. Software support for Ubuntu OS

# 1 Introduction

In this thesis the goal was to find means to implement auto-navigation abilities to a robot through a simulation. There are many different kind of system architecture to provide the navigation such as ROS, ORCA, Player/Player2 and OpenRDK. A comprehensive research work on analyzing different operating systems for the robots was done in thesis by Antti Maula [1]. In his work the criteria to find a suitable operating system was based on five major factors. The first was form of license. The usability and the obligation of a user in implementing the operating system are dependent on the form of license. For example 3-clause BSD license allows the users to use and modify the code freely as long as the license text is preserved in the source code [2]. The second criteria were architectural features. These features represent for example the means for communication between functionalities, devices and sensors. The third feature was expandability. In the expandability perspective, the architectural solution is revised whether it is simple to add new abilities to the system or is the core of the system hard coded. The fourth criteria was documentation. The amount and the quality of the documentation are vital for reusing codes. The fifth and the final criteria was the amount of users and developers involved in the use of system. This affects on the viability of the system as the users and the developers are the essence in sustaining it. With continual improvement of documentation and support gives new users the means and the opportunity to implement the system in to their own work. Base on the Maula’s (2010) research the best candidates were ROS and ORCA operating systems. After further investigation it was noticed that ORCA system was not updated anymore since 2009. Whereas, ROS was still maintained and updated with new features until this day. As a result, it became evident that ROS was to be implemented in this thesis for robot’s navigation features.

# 2 Introduction to Robot Operating System (ROS)

## 2.1 History of ROS

Story of ROS started in mid-2000s as Stanford University was in the phase of creating a system to support STAIR (STandford AI Robot) and Personal robots program (PR). In 2007 Willow Garage, a company located in Menlo Park, California, involved in the development of the system by providing significant resources and thus, contributed to further development of flexible and dynamic software system made for robotics utilization. This also provided more resources and expertise from countless of researches involving in the development. The system was developed under BSD license and gradually has attracted more experts in implementing the system. Over time, it has become a widely-used platform among researches of robotics community. In 2013 the maintenance and the core development of ROS was transferred to Open Source Robotics Foundation and is still running until this day. Currently ROS is implemented by tens of thousands of users all around the world varying from hobby projects to large scale industrial automation system [7].

## 2.2 Structure of ROS

The ROS Structure consists of three different level concepts which are

1) The Filesystem level

2) The Computational Graph level

3) The Community Level

The Filesystem level is the minimum level of creating a necessary building block to operate a function element in ROS (Figure 1). The function element consists of folder structures and files to operate. Each folder has its own functionalities which is composed of six different types [3] that are:

Packages – A package is the core building block of ROS. The package provides the minimum content such as configuration file, nodes and libraries to provide a program inside ROS.

Manifests – A manifest introduces necessary information of the package such as licenses, dependencies and compiler flags.

Stacks – When numerous of packages are combined together and interacting, it forms a stack.

Stack manifest – A stack manifest provides necessary information of the stack such as dependencies and licenses.

Message types (msg) – Message type informs the type of message send to other processes

Services types (srv) – ROS' services define the data structure for responding request and respond.

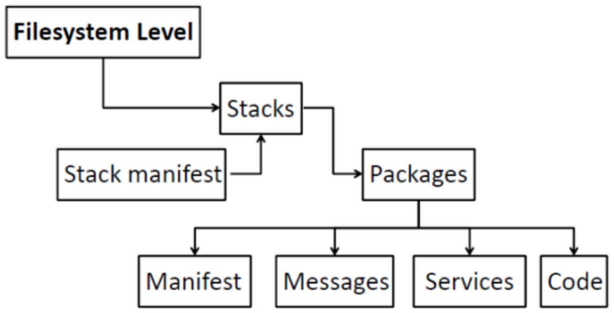


**Figure 1: Filesystem level architect.**

The next level is Computation Graph level where interaction between different system and processes occurs. The basic structure of the level consists of nodes, master, parameter server, services, topics and bags (Figure 2).

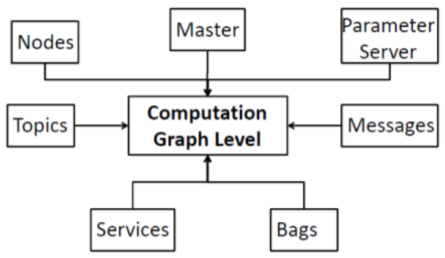


**Figure 2: Computation Graph Level.**

Nodes - The processes consist of nodes where the function’s computation is done. Generally many nodes are created to execute an individual function rather than creating a big node executing all the tasks in the system.

Master - Master handles the name and identification of different nodes. When the nodes are interacting with each other the Master works as search engine in order for the nodes to find one another.

Parameter Server - Parameter server provides the opportunity to configure nodes while the nodes are active and computing with other nodes.

Messages - The communication between nodes is passed through messages where it contains the data for sending information to other nodes. There are many message types such as Int or String. Also own type of messages is possible to create in ROS.

Topics - When the node is sending a data it is also publishing a topic. For another node to receive the data it needs to subscribe to the node’s topic.

Services - If answer or request from a node is needed a service is required as the request can’t be done with topic in this matter. In other words, the services provide the means to interact with different nodes. Each service has to have a unique name.

Bags - In order to store ROS message data and play back, bags can be used for this purpose. It is useful for developing and debugging algorithms where data collection is essential.

The Last level is Community Level. The community level provides ROS resources to separate communities for using software and knowledge through various sources such as distributions, repositories, The ROS Wiki and mailing lists.

Distributions – From ROS distributions collection of stacks can be installed and used. The principle is identical to Linux distributions.

Repositories – Different institutions can create and publish their own robot software component through federated network.

The ROS Wiki – The main source for ROS documentation is from ROS Wiki. In the ROS Wiki anyone can create own account and publish or contribute own documentations, make corrections and create tutorials.

Mailing lists - The main channel for ROS updates and asking question about ROS is through mailing lists.

## 2.3 Navigation Stack

Now that the basic understanding of ROS structure has been established, a comprehensive presentation of auto-navigation feature can be presented. Auto-navigation processing in ROS is implemented in navigation stack where it requires different information in order to make a correct computation towards the desired destination. In this chapter each component is briefly introduced in order to understand what kind of information the component provides and how the information are interacting with each other.

From Figure 3 can been seen features are colored in three different colors white, gray and blue. The white color represents the stack that is already available in ROS for use directly for autonomous navigation. The blue color represents platform that needs to be written a code in order to adapt to ROS and its navigation stack. These platforms are transform frames, odometry information, base controller, sensor information and map server. The grey colors are AMCL (Adaptive Monte Carlo Localization) and map_server. These are optionally provided nodes.

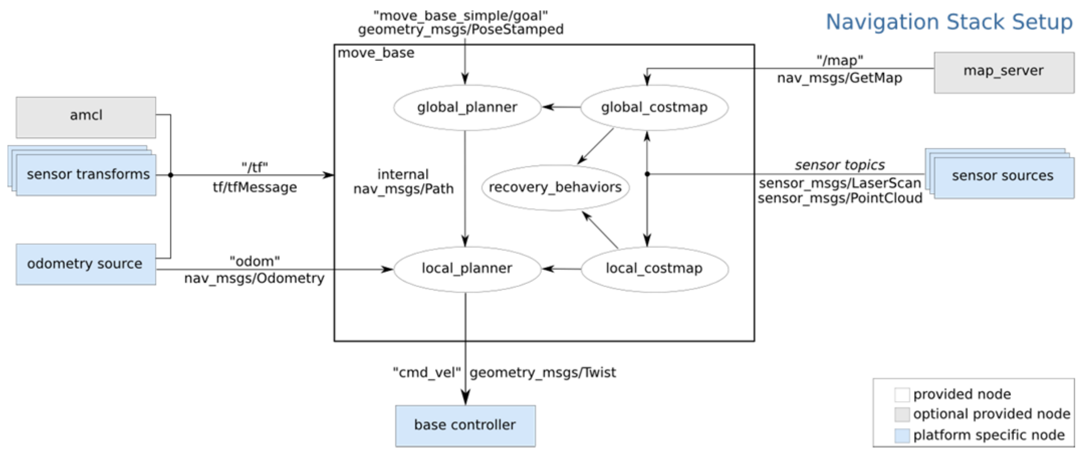


**Figure 3: Structure of Navigation Stack [8].**

Transform configuration provides information of relationship between coordination frames by using TF (Transform Frames). The navigation stack needs to know the position of each wheel, sensor and joints. It is also necessary for the robot to understand the relation between robot's positions compared to the world frame.

Sensor information provides vital data from the laser scanners in the robot for it to avoid obstacles in the world. The laser scan results are published as either sensor_msgs/LaserScan or sensor_msgs/PointCloud messages to the navigation stack.

In order to get an accurate location of the robot after each movement, an odometry information is needed. The odometry information provides velocity data for the navigation stack.

To make the robot move, base controller is needed to receive velocity data and convert it into motor commands for a mobile base.

Map server is not necessary to execute navigation stack but it provides the tool to create a map using odometer and laser sensor data [3].

## 2.4 Gmapping and SLAM

Gmapping package in ROS provides the tools to create a 2-D base map by using laser and odometry data. The method is called SLAM (Simultanous Localization and Mapping). SLAM algorithm creates a map of an unknown environment by performing localization in the area. After the unknown area has been mapped and a robot knows its position related to the map it can perform route planning and navigation. As a result, SLAM algorithm is a vital component for performing auto-navigation for the robot. The laser needs to be equipped with a stationary and horizontally-mounted laser-range-finder [15]. SLAM is also important function for avoiding obstacles along robot's path [18]. A thesis work by Hjelmare and Rangsjö (2012) summarizes that the SLAM algorithm consists of five vital steps [19]:

1) Data Acquisition: Measurements from the sensors such as video camera or laser scanners are collected.

2) Feature extraction: distinctive and recognizable keypoints and features are selected from the data base.

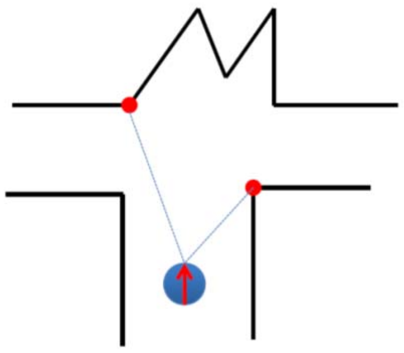
3) Feature association: Keypoints and features from a previous measurements are associated with the most recent keypoints and features.

4) Pose estimation: The relative transition between keypoints and features, and the position of a robot is utilized to estimate the new pose of the robot.

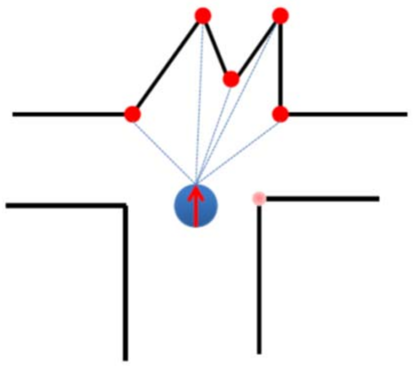
5) Map adjustment: Based on the new pose and the equivalent measurements the map is updated accordingly.

The repetition of the five steps are constantly updated and based on the results the map is created. According to Hjelmare and Rangsjö (2012) The robot first receives an input data from scan results and its position. The keypoints and features are extracted from the data which in this case can be a corner, an edge or a dot in a protruding color.

An illustration from Figure 4a can be seen that the robot identifies two features in the area and saves in a data base consist of all the observed features during the robot's journey. In figure Figure 4b the robot proceeds forward and again identifies new features in the area. At this point, the robot goes through the recorded data base and seeks for the same features it observed. A new feature is added to the data base and the matched feature is used as the estimation of the robot's new pose by measuring transition in the distance and angle to the old features. The new pose estimation is used for adjusting the features it has observed.



b) At time $t_0$, the robot observes features in the area and records its position.



a) At time $t_1$, the robot moves forward and observes new features. One old feature is out of sight and four new features are observed and added.

**Figure 4: Sample of SLAM process.**

## 2.5 Personal Robot TurtleBot

TurtleBot is a personal robot that was created by Tully Foote and Melonee Wise at Willow Garage [16]. The TurtleBot has wide of sensors installed on a mobile base which enables for the robot to navigate around its vicinity based on the ROS. The hardware consists of the following parts [12]. Mobile base and power board provides 3000 mAh Ni-MH Battery Pack, 150 degrees/seconds single axis gyro and 12V 1.5 Amp software enabled power supply for powering the 3D sensor Microsoft Kinetic. Computer for operating the robot with ROS is based on ASUS 1215N laptop with the specs of Intel Atom D525 dual core processor, 2GB Memory, Nvidia Ion Graphics processor and Internal hard drive of 250 GB. 3D sensor is provided via Microsoft Kinect Sensor. TurtleBot's hardware consists of TurtleBot structure, TurtleBot module plate and Kinect mounting set.

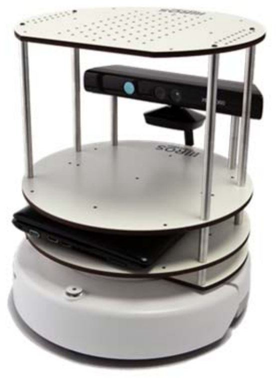


**Figure 5: TurtleBot [12].**

The TurtleBot costs $1200 [13]. However, TurtleBot's operation can be tested through a simulation in ROS with Gazebo simulator and also concurrently with Rviz 3D visualization tool. This opportunity provides the most convenient way to test the TurtleBot's features.

## 2.6 Gazebo Simulator

Gazebo is simulation software collaborated to ROS that provides 3D dynamic simulator (Figure 6). This enables to simulate efficiently and accurately robot’s operation in complex outdoor and indoor environments. A typical use of Gazebo is to test robotics algorithms, robot designs and executing testing in realistic scenarios [14].

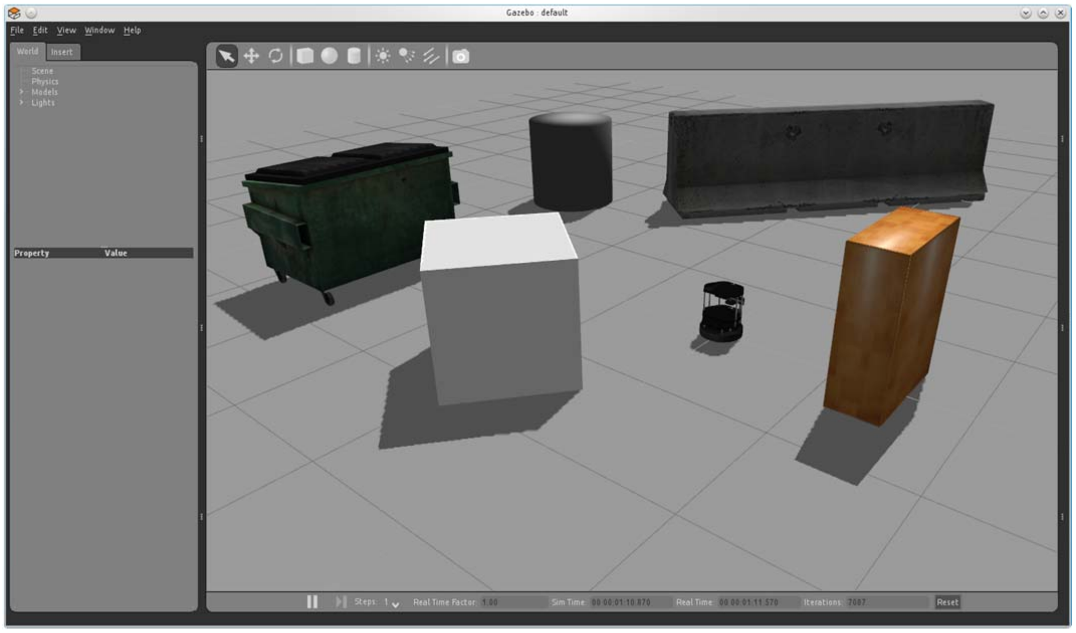


**Figure 6: TurtleBot in Gazebo simulation [15].**

## 2.7 Rviz

Rviz is a 3D visualization tool that provides a convenient way to view data for example from stereo cameras, 3D laser or Kinect sensor. In other words, Rviz helps to visualize sensors' data in a modeled world [3].

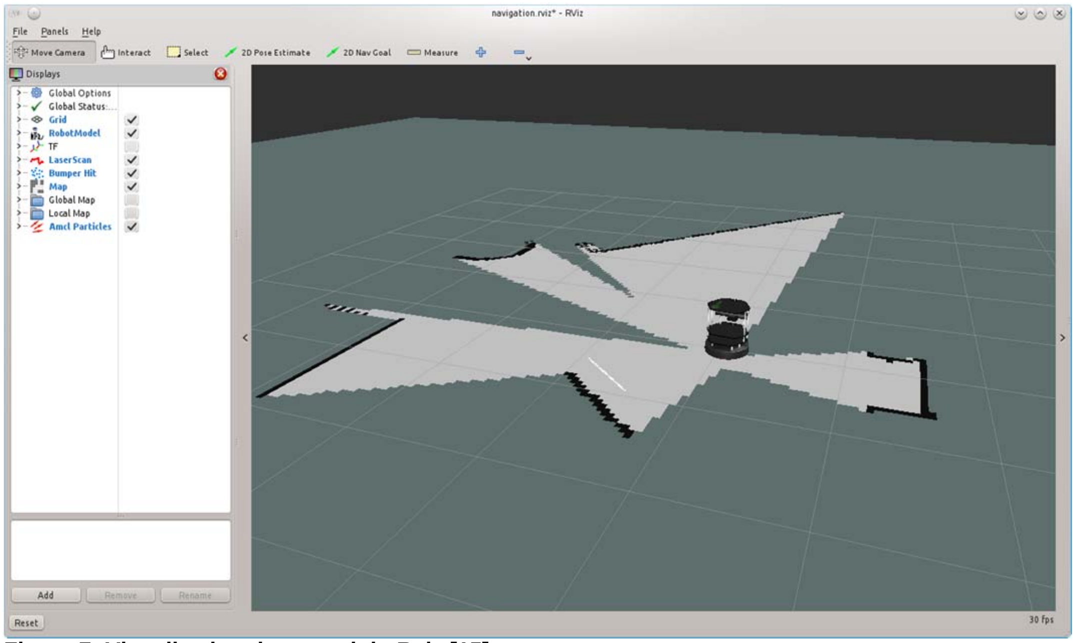


**Figure 7: Visualization data result in Rviz [15].**

# 3 Results

## 3.1 Installing ROS

Installing ROS to Ubuntu operating system was a straightforward process. ROS.org-website provided clear step-by-step guide how to perform installation [4]. At the time of making this thesis, the version of the used Ubuntu was 14.04 LTS. As for the ROS, the latest distribution was codenamed Indigo that was supported by Ubuntu 14.04.

The installation of ROS was executed via terminal software in Ubuntu where different syntaxes were written to initiate the installation. The installation was started by setting the source list in order for the computer to accept software packages from 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'
```

Next step was to set up the keys.

```
wget https://raw.githubusercontent.com/ros/rosdistro/master/ros.key -O - | sudo apt-key
add -
```

When the source list and the set up keys were defined the process of downloading and installing ROS was executed. First was to make sure that the latest Debian package was installed.

```
sudo apt-get update
```

Next step was to download ROS installation package. There are three different options to download the package. First and the recommended one is to download a full-installation.

```
sudo apt-get install ros-indigo-desktop-full
```

This included 2D/3D-simulation programs, navigation and 2D/3D perception, 3D-visualization tool Rviz, GUI development tool called Rqt and robot-generic libraries.

The second option was a basic desktop installation which only included Rqt, Rviz and robot-generic libraries.

```
sudo apt-get install ros-indigo-desktop
```

The third option was ROS-barebone installation which included only ROS package, build and communication libraries.

```
sudo apt-get install ros-indigo-ros-base
```

For this thesis the full desktop installation was chosen. After the installation package was chosen and installed, rosdep was needed to be initialized in order to enable a straightforward installation of system dependencies and to also run some core components in ROS.

```
sudo rosdep init

rosdep update
```

Every time a new shell was launched it was recommended to make ROS add environment variables to bash session every time. This was done by initiating the command.

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

source ~/.bashrc
```

It was also recommended to install a command-line tool, rosinstall, as it was frequently used. Rosinstall enabled to download easily numerous source trees for ROS package with a single command. The syntax for downloading Rosinstall was given.

```
sudo apt-get install python-rosinstall
```

After going through the installation guide, ROS was prepared and ready to be use in the Ubuntu OS.

## 3.2 Installing TurtleBot simulation

Implementation of auto-navigation in ROS was tested with TurtleBot personal robot in a simulated world. Prior launching and testing with the simulation, it was checked that the package installation for TurtleBot was installed [11]. The package can be installed from the terminal by giving the command.

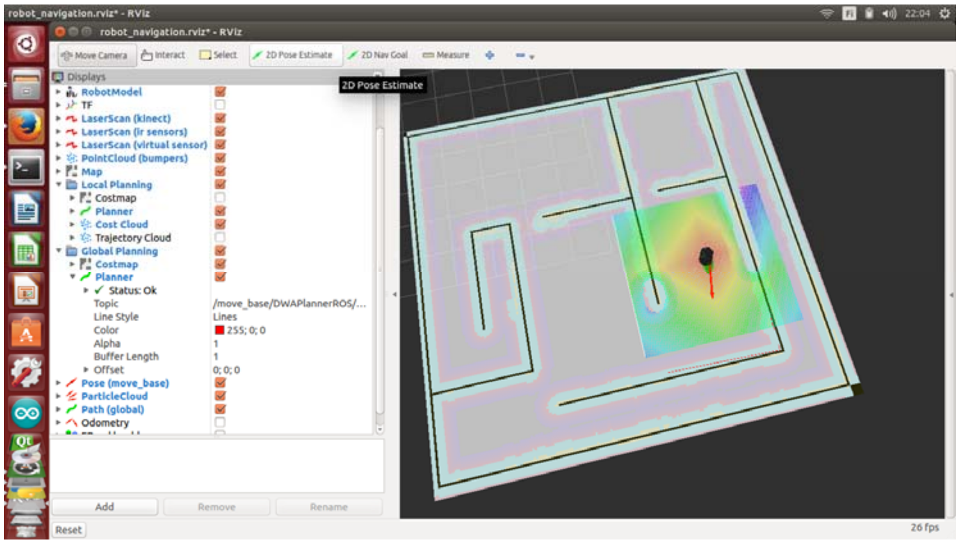
```
sudo apt-get install ros-indigo-turtlebot ros-indigo-turtlebot-apps ros-indigo-turtlebot-interactions ros-indigo-turtlebot-simulator ros-indigo-kobuki-ftdi ros-indigo-rocon-remocon ros-indigo-rocon-qt-library ros-indigo-ar-track-alvar-msgs
```

## 3.3 Testing TurtleBot simulation

The simulation with the TurtleBot was executed by giving the command-line.

```
roslaunch turtlebot_stage turtlebot_in_stage.launch
```

The simulation started by opening three different windows. First window was 3D visualization program, Rviz, for observing all the possible data that the robot receives and creates (Figure 8).



Status: Ok
Topic /move_base/DWAPlannerROS/...
Line Style Lines
Color 🟥 255; 0; 0
Alpha 1
Buffer Length 1
Offset 0; 0; 0

Figure 8: Rviz software for 3D visualization.

The second window, Stage, is a simulation tool part of Player Project. Player Project provides software tools for robot and sensor applications. From Figure 9 can be seen the view of Stage in action where a robot's visual view and trail of its path is visible.

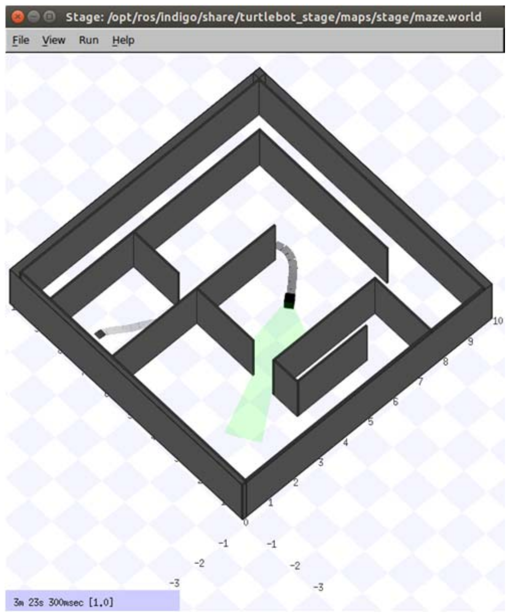

Stage: /opt/ros/indigo/share/turtlebot_stage/maps/stage/maze.world
File View Run Help
3m 23s 300msec [1.0]

**Figure 9: View of Stage simulator.**

A map called Maze was included in the TurtleBot simulation. When the simulation was started the robot was located in the spot according to Figure 10.

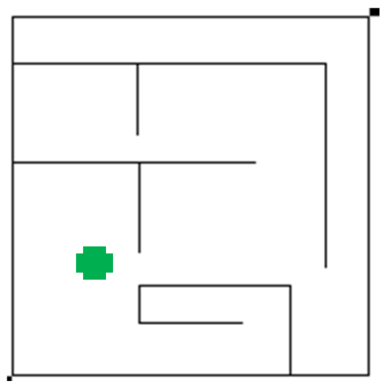


**Figure 10: Maze map where TurtleBot robot navigates. The initial location of TurtleBot in a green spot.**

Before commanding the robot to navigate to a desired location, a setting pose for the robot in the world was required. A 2D pose estimation button was chosen and clicked on the robot in the map while dragging in the direction where the TurtleBot’s head was pointing (Figure 11).

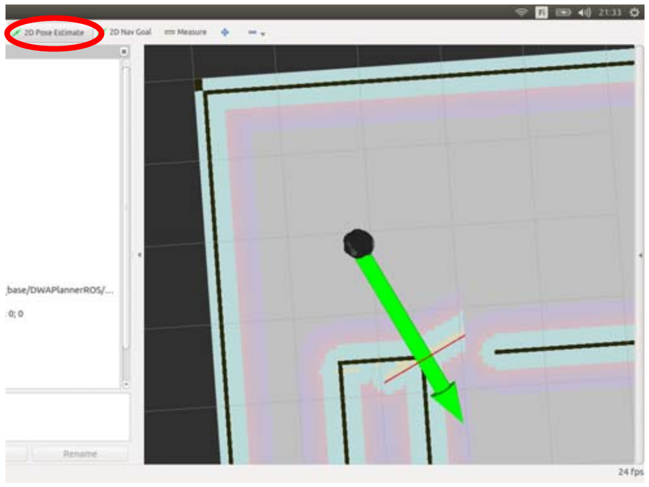


**Figure 11: 2D Pose Estimation with the green arrow indicating the direction of the robot.**

After defining the location of the robot in the map, the TurtleBot was ready to plan the route autonomously through its environment. In order to give a command to move, the 2D Nav button was chosen and clicked on the point in the map while dragging to the direction where the TurtleBot should point at the end of the route. In addition, Costmap, Planner and Cost Cloud visualization were chosen in Rviz to observe how the robot managed to auto-navigate (Figure 12).

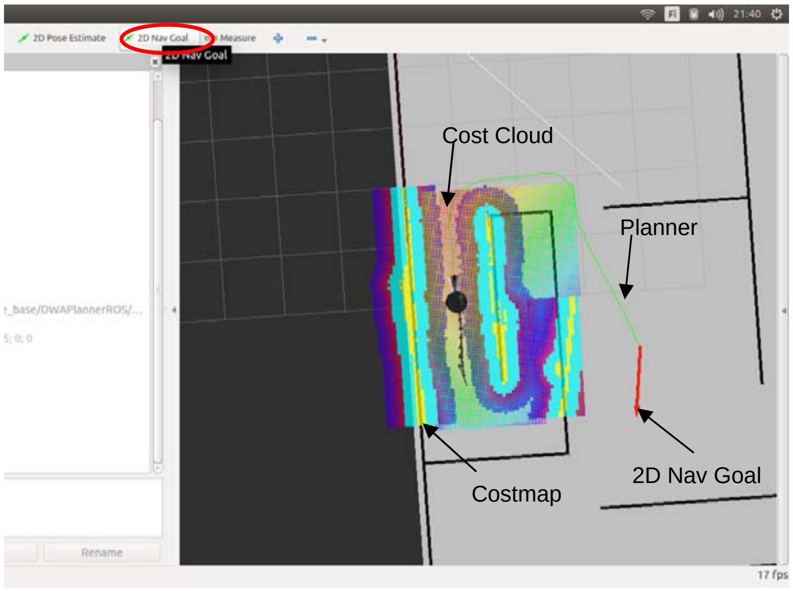


**Figure 12: 2D Nav Goal with visualizations activated.**

After giving the approximate location of the robot and commanding it to a goal point, the robot started to execute the navigation and moved to the destination point. A square field appeared around the robot, 4x4 meters, to indicate the Cost Cloud. It seemed that from a color index of the field it showed the most optimal direction for the TurtleBot to reach the location. Apparently, Red is the most optimal index and blue is the most least optimal. Also the planned route appeared all the way from the Turtle-Bot's location till the end point of the goal when the Planner was activated. The obstacles such as walls in this case appeared through Costmap visualization and seemed to indicate with a yellow color to visualize the wall.

## 3.4 Testing TurtleBot simulation in an unknown environment

Like in the previous paragraph, the simulation was done to a known map area. However a typical case for the robot is that it does not have an idea of its area so it cannot yet auto-navigate freely. With the use of the robot’s laser and odometry data (SLAM method) it can calculate and create a map based on ROS’ tool gmapping algorithm.
The gmapping was tested with a readymade simulation. The simulation was launched by first setting up the setup source in the command-line terminal.

```
$ source /opt/ros/indigo/setup.bash
```

After that, the Gazebo simulation was started with a test world called willowgarage.world.

```
$ roslaunch turtlebot_gazebo turtlebot_world.launch world_file:=worlds/willowgarage.world
```

This simulation created a world where the TurtleBot can roam around the indoor environment (Figure 13).

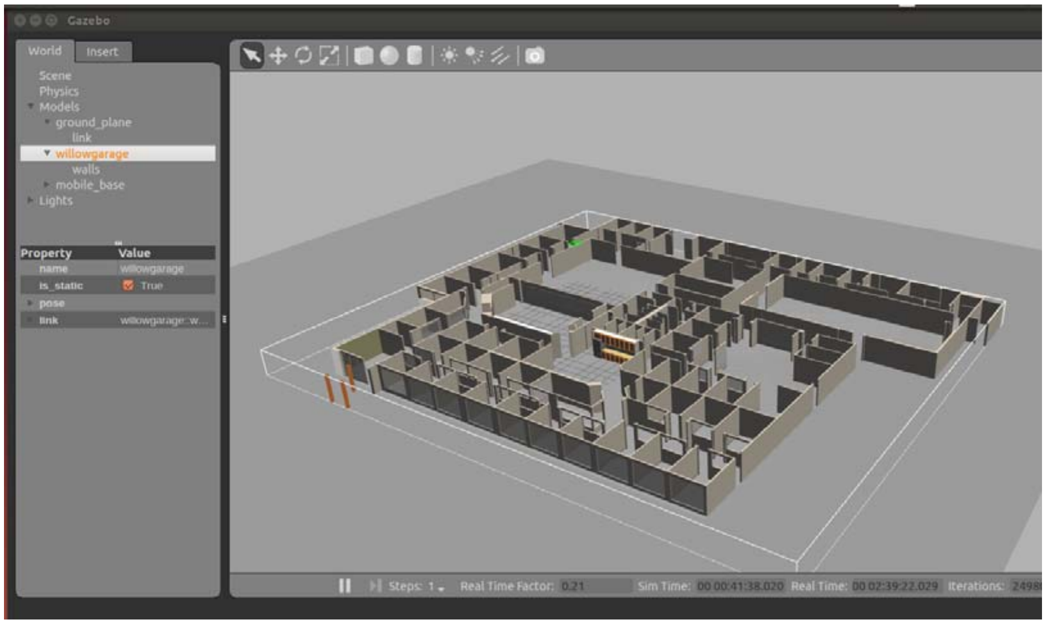
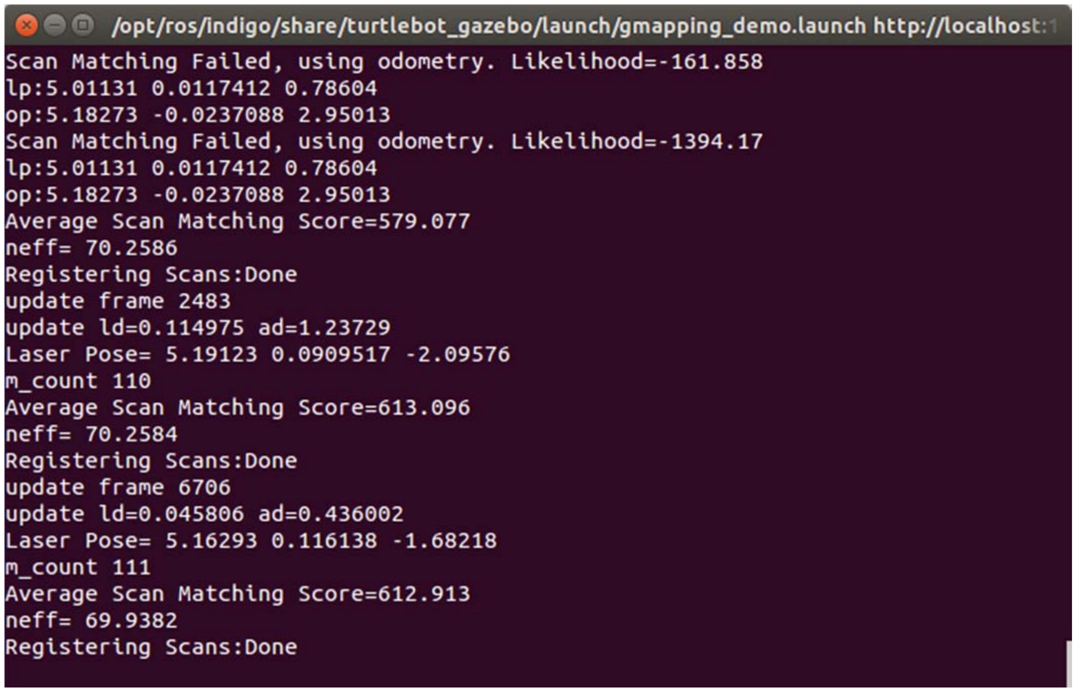


**Figure 13: Gazebo simulation world.**

Next step was to create a map of the indoor environment by launching gmapping tool. The command-line in the terminal was given.

```
$ roslaunch turtlebot_gazebo gmapping_demo.launch
```

The terminal initiated status information of gmapping's creation process (Figure 14).



```
/opt/ros/indigo/share/turtlebot_gazebo/launch/gmapping_demo.launch http://localhost:1
Scan Matching Failed, using odometry. Likelihood=-161.858
lp:5.01131 0.0117412 0.78604
op:5.18273 -0.0237088 2.95013
Scan Matching Failed, using odometry. Likelihood=-1394.17
lp:5.01131 0.0117412 0.78604
op:5.18273 -0.0237088 2.95013
Average Scan Matching Score=579.077
neff= 70.2586
Registering Scans:Done
update frame 2483
update ld=0.114975 ad=1.23729
Laser Pose= 5.19123 0.0909517 -2.09576
m_count 110
Average Scan Matching Score=613.096
neff= 70.2584
Registering Scans:Done
update frame 6706
update ld=0.045806 ad=0.436002
Laser Pose= 5.16293 0.116138 -1.68218
m_count 111
Average Scan Matching Score=612.913
neff= 69.9382
Registering Scans:Done
```

**Figure 14: Status information of map creation process in gmapping.**

The visualization of the 2-D map progress was viewed from Rviz. The command-line in the terminal was executed.

```
$ roslaunch turtlebot_rviz_launchers view_navigation.launch
```

From Figure 15 can be seen the result of gmapping when the TurtleBot roamed around the environment.

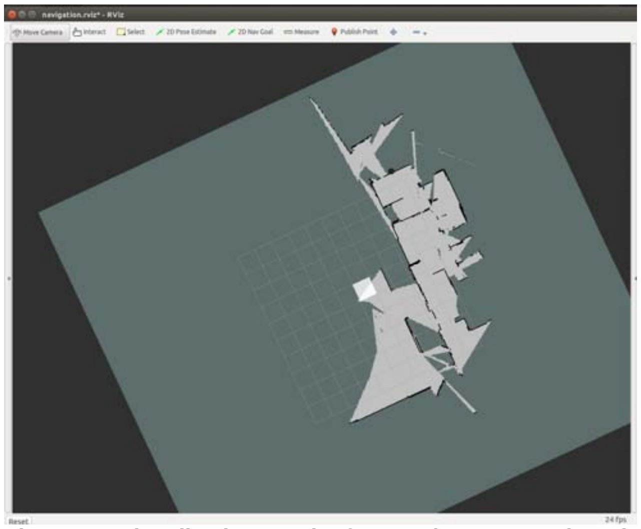
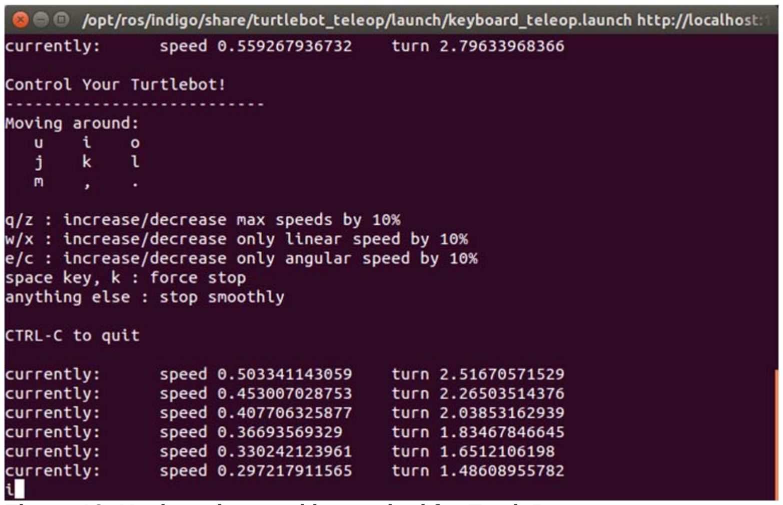


**Figure 15: Visualization result of gmapping process in Rviz.**

In order to make the TurtleBot move manually around the environment a keyboard teleoperation was needed. The command-line in the terminal was executed.

```
$ roslaunch turtlebot_teleop keyboard_teleop.launch
```



```
/opt/ros/indigo/share/turtlebot_teleop/launch/keyboard_teleop.launch http://localhost:
currently:      speed 0.559267936732    turn 2.79633968366

Control Your Turtlebot!
---------------------------
Moving around:
   u    i    o
   j    k    l
   m    ,    .

q/z : increase/decrease max speeds by 10%
w/x : increase/decrease only linear speed by 10%
e/c : increase/decrease only angular speed by 10%
space key, k : force stop
anything else : stop smoothly

CTRL-C to quit

currently:      speed 0.503341143059    turn 2.51670571529
currently:      speed 0.453007028753    turn 2.26503514376
currently:      speed 0.407706325877    turn 2.03853162939
currently:      speed 0.36693569329     turn 1.83467846645
currently:      speed 0.330242123961    turn 1.6512106198
currently:      speed 0.297217911565    turn 1.48608955782
i
```

**Figure 16: Keyboard control in terminal for TurtleBot.**

The basic movement in the keyboard was forward with i-keyboard, left with j-keyboard, right with l-keyboard and reverse with ,-keyboard. Force stop was k-keyboard.

The created map from gmapping was saved by giving the command-line in the terminal.

```
$ rosrun map_server map_saver -f <your map name>
```

After the map was saved, it can be tested for auto-navigation. The map file is saved as YAML-file. In order to launch the created map, the command-line was given in the terminal.

```
roslaunch turtlebot_gazebo amcl_demo.launch map_file:=<full path to your map YAML file>
```

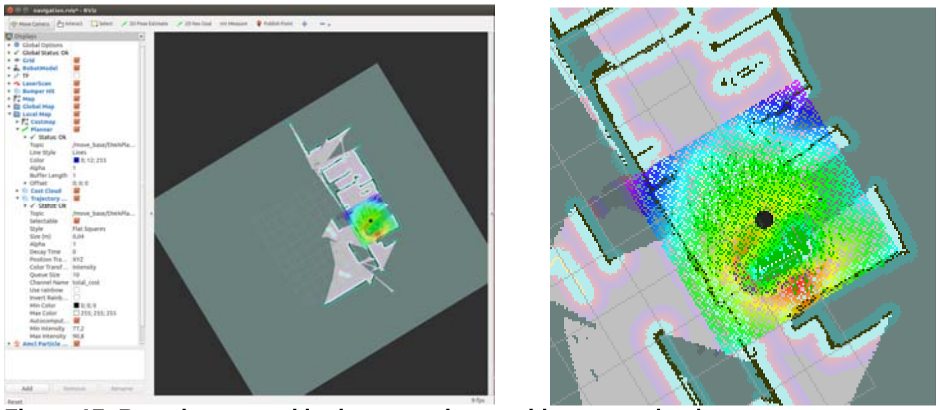


**Figure 17: Roaming around in the created map with auto-navigation.**

The same control procedure for the TurtleBot was applied to roam around the created map like in the chapter 3.3. First, the 2D pose estimated was defined for the TurtleBot. Secondly, the TurtleBot was ordered to move around the created map with 2D nav goal button. Finally, the planned route appeared all the way from the TurtleBot's location till the end point of the goal.

# 4 Conclusion

Installing and testing navigation stack through the simulation was straight forward execution as the instructions from site tutorials were clear. Testing the navigation stack with TurtleBot in simulated world was also clear by following the tutorials.

By simply clicking first 2D Pose Estimate button to the TurtleBot and then click 2D Nav Goal button and the desired location in the map made the TurtleBot immediately auto-navigated to the destination. It is important to define the 2D Pose Estimation directly on the robot and in the correct direction as failing to do so may cause problem with the navigation and in the end cause failing to find the destination. Problem with the navigation can also occur when the destination is far away and the path is narrow. This will result in failure to create and find the path for the robot to navigate. The counter measurement for this purpose appeared to be rotate recovery behavior operation. The robot rotated twice around to scan the area and searched for a possible route. If the robot still failed to find a route it decided to abort and informed that it could not find valid control. The apparent solution was to give a destination point with smaller steps before giving the final goal of the destination. After giving the small steps first, the robot managed to find the final destination point. Martinez and Fernandez (2013) pointed out that auto-navigation stack is applicable only to a certain type of robot in order for it to work. The navigation stack can only be implemented to a holonomic-wheeled with a differential drive. In addition, for the robot to observe the environment, a planar laser is needed to create localization and map. The first simulation test of auto-navigation was performed to a readymade map.

However, if a map is not readily available there is a solution based on gmapping with SLAM method. In SLAM method, TurtleBot roams around the unknown area while simultaneously scans the area by using robot's odometry and laser sensor data. The collected data is sent to map server stack and gmapping package in the ROS to form a map. In a research work by Afanasyev, Sagitov and Magid (2015) summarized that map creation based on SLAM in ROS with Gazebo simulation and Rviz visualization consisted of the following steps. First, Simulation is started by ROS in Gazebo. Secondly, TurtleBot's movement and sensor measurements are created in Gazebo simulation and the results exported to ROS. Thirdly, the calculation of SLAM mapping and robot localization is calculated in ROS. Finally, simulated data results is visualized and imported in Rviz from ROS.

When the gmapping was tested in chapter 3.4, it was noticed that the tool had difficulties to scan the area when the TurtleBot roamed. The gmapping status informed constantly that scan matching failed. Visually the problem was seen in the Rviz where the TurtleBot could not plot the scanned area and the location of the TurtleBot was also inaccurate as, occasionally, the TurtleBot made sudden teleportation movements in the simulated world. In order to improve the ability to scan the area and also keep track of the TurtleBot's movement, the speed was decreased. This improved the performance and the chance for the TurtleBot to create the area. Nevertheless, this did not entirely remove the difficulties in creating the map. As a result, the map scanning and creating process was slow phase as in each step had to be made sure that the TurtleBot was able to scan and register the area. Another way was to repeat the same route in order to create the map properly. The result of the created map was visualized through Rviz. The scanned area was successfully created if it appeared as a grey area in the Rviz. Through numerous trial and error methods the map was created and the TurtleBot managed to auto-navigate around the created map.

## 5 References

1 **Maula A., 2010.** Avoimen lähdekoodin kirjastot robotiikassa, sekä niiden soveltaminen työkoneympäristöön.
Helsinki, Finland: Helsinki Metropolia University of Applied Sciences.

2 **Robot Operating System, 2015.** 3-clause BSD license. [online] available at: http://opensource.org/licenses/BSD-3-Clause [Accessed 07 February 2015]

3 **Martinez, A. and Fernandez, E., 2013**. Learning ROS for Robotics Programming
Birmingham: Packt Publishing

4 **Robot Operating System, 2015.** Installation Guide. [online] available at: http://wiki.ros.org/ROS/Installation [Accessed 07 February 2015]

5 **Wiki-ROS, 2015.** Concept [online] available at: http://wiki.ros.org/ROS/Concepts [Accessed 07 February 2015]

6 **WIKI-ROS, 2015.** Message Description Specification. [online] available at: http://wiki.ros.org/msg [Accessed 07 February 2015]

7 **WIKI-ROS, 2015.** History. [online] available at: http://wiki.ros.org/history [Accessed 07 February 2015]

8 **WIKI-ROS, 2015.** Robot setup. [online] available at: http://wiki.ros.org/navigation/Tutorials/RobotSetup [Accessed 07 February 2015]

9 **WIKI-ROS, 2015.** gmapping. [online] available at: http://wiki.ros.org/gmapping [Accessed 07 February 2015]

10 **Robot Operating System, 2015.** ROS Cheat Sheet. [online] available at: http://www.ros.org/news/2015/05/ros-cheatsheet-updated-for-indigo-igloo.html [Accessed 07 February 2015]

11 **WIKI-ROS, 2015.** Turtlebot In Stage Simulator. [online]. available at: http://wiki.ros.org/turtlebot_stage/Tutorials/indigo/Bring%20up%20TurtleBot%20in%20stage [Accessed 07 February 2015]

12 **Willow Garage, 2015.** TurtleBot Hardware Specs. [online] Available at:
https://www.willowgarage.com/turtlebot/specs.
[Accessed 05 Mar 2016]

13 **Saenz A., 2011.** Willow Garage Is Selling An Amateur Level Robot! Say Hi To TurtleBot. [online] available at: http://singularityhub.com/2011/04/19/willow-garage-is-selling-an-amateur-level-robot-say-hi-to-turtlebot-video/. [accessed 12 Feb 2016]

14 **Gazebo, 2016.** Gazebo Overview. [online] available at: gazebosim.org/tutorials?cat=guided_b&tut=guided_b2

15 **Wiki-ROS, 2015.** TurtleBot Gazebo. [online] available at: http://wiki.ros.org/turtlebot_gazebo/Tutorials/indigo/Make%20a%20map%20and%20navigate%20with%20it

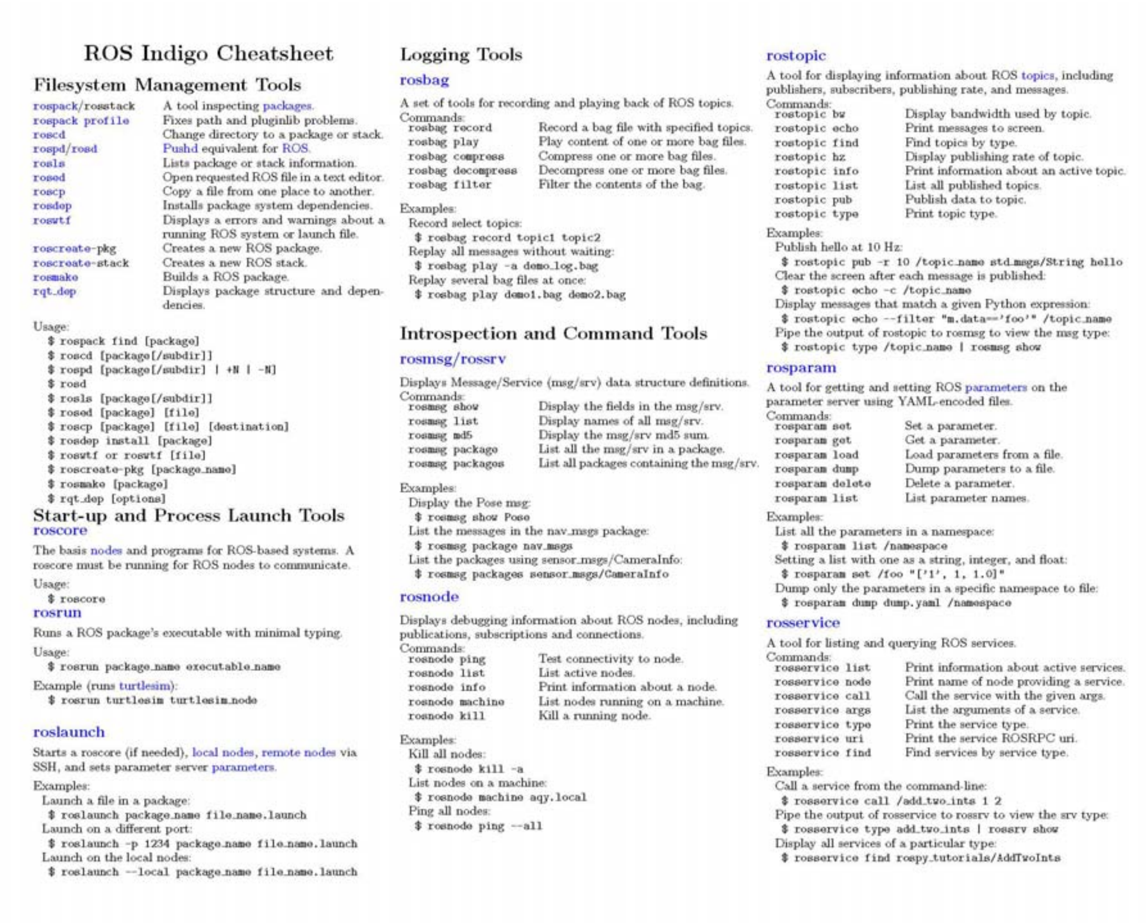
16 **Ackerman E., 2013.** Interview: TurtleBot Inventors Tell Us Everything About the Robot. [online] Available at: http://spectrum.ieee.org/automaton/robotics/diy/interview-turtlebot-inventors-tell-us-everything-about-the-robot.
[Accessed 09 Mar 2016]

17 **Afanasyev I., Sagitov A. and Magid E., 2015.** ROS-based SLAM for a Gazebo-simulated mobile robot in image-based 3D model of indoor environment. [online] Available at: https://university.innopolis.ru/files/ROS-based%20SLAM%20for%20a%20Gazebo-simulated%20mobile%20robot%20in%20image-based%203D%20mod....pdf.
[Accessed 05 Mar 2016]

18 **Kouba A., 2016.** Robot Operating System (ROS). The Complete Reference (Volume 1).
Springer International Publishing AG Switzerland

19 **Hjelmare F. and Rangsjö J., 2012**. Simultaneous Localization And Mapping Using a Kinect™ In a Sparse Feature Indoor Environment.
Linköping, Sweden: Linköpings Universitet

# 6 Appendices



## ROS Indigo Cheatsheet

### Filesystem Management Tools

| | |
|---|---|
| rospack/rosstack | A tool inspecting packages. |
| rospack profile | Fixes path and pluginlib problems. |
| roscd | Change directory to a package or stack. |
| rospd/rosd | Pushd equivalent for ROS. |
| rosls | Lists package or stack information. |
| rosed | Open requested ROS file in a text editor. |
| roscp | Copy a file from one place to another. |
| rosdep | Installs package system dependencies. |
| roswtf | Displays a errors and warnings about a running ROS system or launch file. |
| roscreate-pkg | Creates a new ROS package. |
| roscreate-stack | Creates a new ROS stack. |
| rosmake | Builds a ROS package. |
| rqt_dep | Displays package structure and dependencies. |

Usage:

```
$ rospack find [package]
$ roscd [package[/subdir]]
$ rospd [package[/subdir] | +N | -N]
$ rosd
$ rosls [package[/subdir]]
$ rosed [package] [file]
$ roscp [package] [file] [destination]
$ rosdep install [package]
$ roswtf or roswtf [file]
$ roscreate-pkg [package_name]
$ rosmake [package]
$ rqt_dep [options]
```

### Start-up and Process Launch Tools

#### roscore

The basis nodes and programs for ROS-based systems. A roscore must be running for ROS nodes to communicate.

Usage:

```
$ roscore
```

#### rosrun

Runs a ROS package's executable with minimal typing.

Usage:

```
$ rosrun package_name executable_name
```

Example (runs turtlesim):

```
$ rosrun turtlesim turtlesim_node
```

#### roslaunch

Starts a roscore (if needed), local nodes, remote nodes via SSH, and sets parameter server parameters.

Examples:

Launch a file in a package:
```
$ roslaunch package_name file_name.launch
```
Launch on a different port:
```
$ roslaunch -p 1234 package_name file_name.launch
```
Launch on the local nodes:
```
$ roslaunch --local package_name file_name.launch
```

### Logging Tools

#### rosbag

A set of tools for recording and playing back of ROS topics.

Commands:

| | |
|---|---|
| rosbag record | Record a bag file with specified topics. |
| rosbag play | Play content of one or more bag files. |
| rosbag compress | Compress one or more bag files. |
| rosbag decompress | Decompress one or more bag files. |
| rosbag filter | Filter the contents of the bag. |

Examples:

Record select topics:
```
$ rosbag record topic1 topic2
```
Replay all messages without waiting:
```
$ rosbag play -a demo_log.bag
```
Replay several bag files at once:
```
$ rosbag play demo1.bag demo2.bag
```

### Introspection and Command Tools

#### rosmsg/rossrv

Displays Message/Service (msg/srv) data structure definitions.

Commands:

| | |
|---|---|
| rosmsg show | Display the fields in the msg/srv. |
| rosmsg list | Display names of all msg/srv. |
| rosmsg md5 | Display the msg/srv md5 sum. |
| rosmsg package | List all the msg/srv in a package. |
| rosmsg packages | List all packages containing the msg/srv. |

Examples:

Display the Pose msg:
```
$ rosmsg show Pose
```
List the messages in the nav_msgs package:
```
$ rosmsg package nav_msgs
```
List the packages using sensor_msgs/CameraInfo:
```
$ rosmsg packages sensor_msgs/CameraInfo
```

#### rosnode

Displays debugging information about ROS nodes, including publications, subscriptions and connections.

Commands:

| | |
|---|---|
| rosnode ping | Test connectivity to node. |
| rosnode list | List active nodes. |
| rosnode info | Print information about a node. |
| rosnode machine | List nodes running on a machine. |
| rosnode kill | Kill a running node. |

Examples:

Kill all nodes:
```
$ rosnode kill -a
```
List nodes on a machine:
```
$ rosnode machine aqy.local
```
Ping all nodes:
```
$ rosnode ping --all
```

#### rostopic

A tool for displaying information about ROS topics, including publishers, subscribers, publishing rate, and messages.

Commands:

| | |
|---|---|
| rostopic bw | Display bandwidth used by topic. |
| rostopic echo | Print messages to screen. |
| rostopic find | Find topics by type. |
| rostopic hz | Display publishing rate of topic. |
| rostopic info | Print information about an active topic. |
| rostopic list | List all published topics. |
| rostopic pub | Publish data to topic. |
| rostopic type | Print topic type. |

Examples:

Publish hello at 10 Hz:
```
$ rostopic pub -r 10 /topic_name std_msgs/String hello
```
Clear the screen after each message is published:
```
$ rostopic echo -c /topic_name
```
Display messages that match a given Python expression:
```
$ rostopic echo --filter "m.data=='foo'" /topic_name
```
Pipe the output of rostopic to rosmsg to view the msg type:
```
$ rostopic type /topic_name | rosmsg show
```

#### rosparam

A tool for getting and setting ROS parameters on the parameter server using YAML-encoded files.

Commands:

| | |
|---|---|
| rosparam set | Set a parameter. |
| rosparam get | Get a parameter. |
| rosparam load | Load parameters from a file. |
| rosparam dump | Dump parameters to a file. |
| rosparam delete | Delete a parameter. |
| rosparam list | List parameter names. |

Examples:

List all the parameters in a namespace:
```
$ rosparam list /namespace
```
Setting a list with one as a string, integer, and float:
```
$ rosparam set /foo "['1', 1, 1.0]"
```
Dump only the parameters in a specific namespace to file:
```
$ rosparam dump dump.yaml /namespace
```

#### rosservice

A tool for listing and querying ROS services.

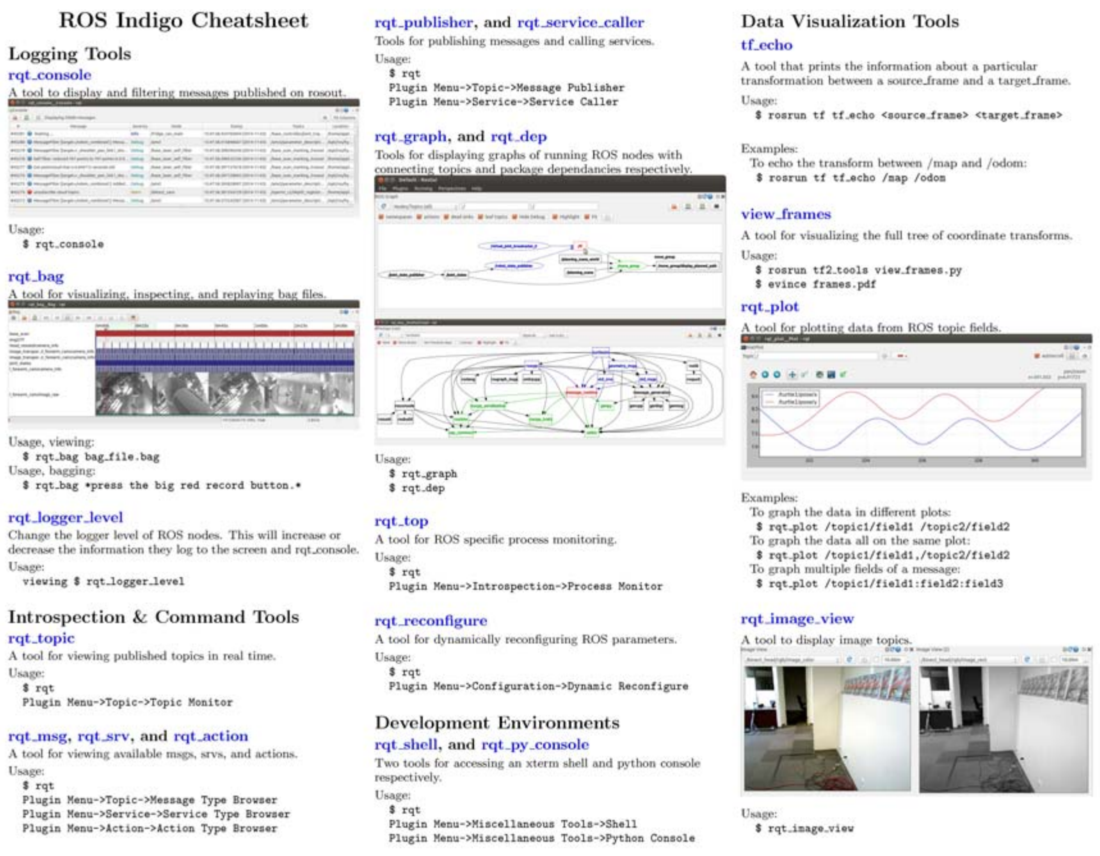
Commands:

| | |
|---|---|
| rosservice list | Print information about active services. |
| rosservice node | Print name of node providing a service. |
| rosservice call | Call the service with the given args. |
| rosservice args | List the arguments of a service. |
| rosservice type | Print the service type. |
| rosservice uri | Print the service ROSRPC uri. |
| rosservice find | Find services by service type. |

Examples:

Call a service from the command-line:
```
$ rosservice call /add_two_ints 1 2
```
Pipe the output of rosservice to rossrv to view the srv type:
```
$ rosservice type add_two_ints | rossrv show
```
Display all services of a particular type:
```
$ rosservice find rospy_tutorials/AddTwoInts
```

**Appendix 1: ROS cheat sheet page 1 [10].**



ROS Indigo Cheatsheet

Logging Tools

rqt_console

A tool to display and filtering messages published on rosout.

Usage:
```
$ rqt_console
```

rqt_bag

A tool for visualizing, inspecting, and replaying bag files.

Usage, viewing:
```
$ rqt_bag bag_file.bag
```
Usage, bagging:
```
$ rqt_bag *press the big red record button.*
```

rqt_logger_level

Change the logger level of ROS nodes. This will increase or decrease the information they log to the screen and rqt_console.

Usage:
```
viewing $ rqt_logger_level
```

Introspection & Command Tools

rqt_topic

A tool for viewing published topics in real time.

Usage:
```
$ rqt
Plugin Menu->Topic->Topic Monitor
```

rqt_msg, rqt_srv, and rqt_action

A tool for viewing available msgs, srvs, and actions.

Usage:
```
$ rqt
Plugin Menu->Topic->Message Type Browser
Plugin Menu->Service->Service Type Browser
Plugin Menu->Action->Action Type Browser
```

rqt_publisher, and rqt_service_caller

Tools for publishing messages and calling services.

Usage:
```
$ rqt
Plugin Menu->Topic->Message Publisher
Plugin Menu->Service->Service Caller
```

rqt_graph, and rqt_dep

Tools for displaying graphs of running ROS nodes with connecting topics and package dependancies respectively.

Usage:
```
$ rqt_graph
$ rqt_dep
```

rqt_top

A tool for ROS specific process monitoring.

Usage:
```
$ rqt
Plugin Menu->Introspection->Process Monitor
```

rqt_reconfigure

A tool for dynamically reconfiguring ROS parameters.

Usage:
```
$ rqt
Plugin Menu->Configuration->Dynamic Reconfigure
```

Development Environments

rqt_shell, and rqt_py_console

Two tools for accessing an xterm shell and python console respectively.

Usage:
```
$ rqt
Plugin Menu->Miscellaneous Tools->Shell
Plugin Menu->Miscellaneous Tools->Python Console
```

Data Visualization Tools

tf_echo

A tool that prints the information about a particular transformation between a source_frame and a target_frame.

Usage:
```
$ rosrun tf tf_echo <source_frame> <target_frame>
```

Examples:
To echo the transform between /map and /odom:
```
$ rosrun tf tf_echo /map /odom
```

view_frames

A tool for visualizing the full tree of coordinate transforms.

Usage:
```
$ rosrun tf2_tools view_frames.py
$ evince frames.pdf
```

rqt_plot

A tool for plotting data from ROS topic fields.

Examples:
To graph the data in different plots:
```
$ rqt_plot /topic1/field1 /topic2/field2
```
To graph the data all on the same plot:
```
$ rqt_plot /topic1/field1,/topic2/field2
```
To graph multiple fields of a message:
```
$ rqt_plot /topic1/field1:field2:field3
```

rqt_image_view

A tool to display image topics.

Usage:
```
$ rqt_image_view
```

**Appendix 2: ROS cheat sheet page 2 [10].**