Robotic Engineering 2: ROS-Turtlebot Motion Control and Navigation

11th May, 2015

AK ASSAD
MASHRUF CHOWDHURY
YANIK PORTO

Robotics Engineering 2: ROS-Turtlebot Motion Control and Navigation

AK Assad   Mashruf Chowdhury   Yanik Porto

May 11, 2015

Contents

1 Part 1 - Motion Control

1.1 Understanding the basics

- Nodes: A node is an executable that uses ROS to communicate with other nodes.
- Messages: ROS data type used when subscribing or publishing to a topic.
- Topics: Nodes can publish messages to a topic as well as subscribe to a topic to receive messages.
- Master: Name service for ROS (i.e. helps nodes find each other)
- rosout: ROS equivalent of stdout/stderr, This is always running as it collects and logs nodes' debugging output.
- roscore: Master + rosout + parameter server (parameter server will be introduced later)

1.2 Levels of Motion Control

Controlling a mobile robot can be done at a number of levels and ROS provides methods for most of them.
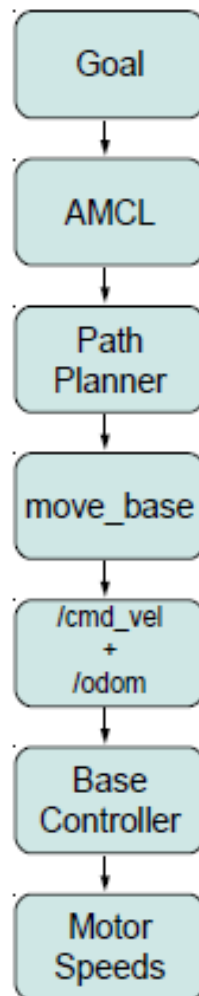
Levels of motion control

1.2.1 Motors, Wheels and Encoders

Turtlebot uses **Encoders** to drive it's motors or wheels. Encoder registers certain number of ticks per revolution for a wheel. Knowing the diameter and distance between the wheels, from these ticks we can find distance traveled by the robot. To compute speed, these values are simply divided by the time interval between measurements. This internal motion data is known as odometry . Due to Environmental and other errors this data is not generally accurate. We need to use other motion data sources to get better estimation.

1.2.2 Motor Controllers and Drivers

At the lowest level of motion control, a **driver** for the robot's motor controller turns the wheels at a desired speed, usually using internal units such as **encoder ticks per second** or a percentage of max speed.

1.2.3 The ROS Base Controller

At this level of abstraction, the desired speed of the robot is specified in real-world units such as meters and radians per second. It also commonly employs some **PID control** (Proportional Integral Derivative), which in layman terms generally does it's best to move the robot in the way we have requested.

The driver and PID control are combined inside a single ROS node called **base controller**.

The base controller node typically publishes odometry data on the **/odom** topic and listens for motion commands on the **/cmd_vel** topic.

Also a transform from the **/odom** frame to the base frame—either **/base_link** or **/base_footprint (Turtlebot)** is done depending on the robot types.

Once we have our **base controller with a ROS interface** our programming can focus purely on the desired linear and angular velocities in real-world units.

1.2.4 Frame-Base Motion using the move_base ROS Package

The **move_base** package is a very sophisticated path planner and combines **odometry data** with both **local and global cost maps** when selecting a path for the robot to follow. We discuss more about **move_base** in Navigating with Path Planning section.

1.2.5 SLAM using the gmapping and amcl ROS Packages

At an even higher level, ROS enables our robot to create a map of its environment using the SLAM gmapping package.

Once a map of the environment is available, ROS provides the amcl package (adaptive Monte Carlo localization) for automatically localizing the robot based on its current scan and odometry data.

1.2.6 Semantic Goals

Finally, at the highest level of abstraction, motion goals are specified semantically such as "go to the kitchen and bring me a beer", or simply, "bring me a beer". These are generally segmented into smaller tasks and passed to lower levels.

1.3 Setting up the Network

The primary requirement for getting your TurtleBot running is connecting to a network so that we can operate it remotely. After both TurtleBot and Workstation is connected to a network, it is time to set ROS_MASTER_URI and ROS_HOSTNAME. The ROS_MASTER_URI tells the rest of the nodes at which address they can find the ROS master.

Here we setup the TurtleBot netbook as a ROS Master and our Workstation as a ROS node. Before this is done, we need to determine the IP address of your TurtleBot netbook and Workstation. This can be found by typing "ifconfig" in a terminal. Our IP address are found under wlan0, and it's the numbers proceeding "inet addr:".

```
> echo export ROS_MASTER_URI=http://IP_OF_TURTLEBOT:11311 >> ~/.bashrc
> echo export ROS_HOSTNAME=IP_OF_WORKSTATION >> ~/.bashrc
```

For ROS to operate you need to have an instance of roscore running on the master. We type the following command in the master:

```
$ roscore
```

These commands will add the export lines to the system bashrc file so they will run on every new terminal instance. This could also be done using an editor with command like $gedit \sim /.bashrc$ and saving the IP addresses there.

1.4 Testing with the fake and the real Turtlebot

The programs can be tested both in a simulated environment and in the real robot. However, when testing the real robot using the current configuration of the kobuki base in the turtlebot, it is necessary to remap the topic **/mobile_base/commands/velocity** to /**cmd_vel**. In order do this, a launch file is created to remap this topic, in fact in every launch file we changed the topic like this.

1.5 Goal 1 : Basic Motion of Mobile Base

We can do this in two way - from command prompt or by using a node. We see both way -

1.5.1 From command prompt

ROS uses the **Twist message** type for publishing motion commands to be used by the base controller. While we could use almost any name for a topic, it is usually called **/cmd_vel** which is short for "command velocities". The base controller node subscribes to the **/cmd_vel** topic and translates **Twist messages** into motor signals that actually turn the wheels.

We can see the components of a **Twist message** using the following command:

$rosmsg show geometry_msgs/Twist

The Twist message is composed of two sub-messages with type Vector3, one for the x, y and z linear velocity components and another for the x, y and z angular velocity components. Linear velocities are specified in meters per second and angular velocities are given in radians per second.

By the following command we say the robot to go in linear direction 0.15 meter/s and at the same time change angular direction (z axis) clockwise 0.4 rad/s. For example if we wanted to rotate a full revolution per 2 second we had to use 3.14 as z value.

```
$roslaunch turtlebot_le2i remap_rplidar_minimal.launch (Bring up turtlebot)
$rostopic pub -r 10 /cmd_vel geometry_msgs/Twist '{linear: {x: 0.15, y: 0, z: 0},
    angular: {x: 0, y: 0, z: -0.4}}'
```

We initialize the launch file and Publish **geometry_msgs/Twist** message in the **/cmd_vel** topic using "rostopic" command. Here **-r** is an option by which we say to do 10 repetitions.

1.5.2 By using a node

To do this we run a python script from ROS-by-Example (rbx1) packages. At first we go to our **/ros_workspace** directory and clone the repo and make the package using the following commands:

```
$git clone https://github.com/pirobot/rbx1.git
$cd rbx1
$rosmake
$rospack profile
```

From command prompt we just use one command at a time. But we want to do some sequential tasks, so it is better to give commands again and again then to put them in a source code file or node.
We use timed Twist commands to move the robot forward a certain distance, rotate 180 degrees, then move forward again for the same time and at the same speed where it will hopefully end up where it started. Finally, we will rotate the robot 180 degrees one more time to match the original orientation.

Algorithm 1: Time and Speed based out and back

```
rate = 50
goal_distance = 1.0 m
linear_speed = 0.2m/s
linear_duration = goal_distance / linear_speed;

angular_speed = 1.0
goal_angle = pi
angular_duration = goal_angle / angular_speed;
```

```
A: Repeat step 1 to 6 twice

  1. ticks = linear_duration / rate
  2. while(t < ticks)
     robot will walk 1 meter straight
  3. robot will sleep 1 cycle

  4. ticks = goal_angle * rate
  5. while(t<ticks)
     robot rotates 180*
  6. robot will sleep 1 cycle

B: Program Terminates
```

This is a **Time and Speed** based approximation, which we shall see will not be as accurate as expected. Here are the code segments to do it:

We ssh into the turtlebot and run the follwing scripts on Turtlebot:

```
$roslaunch turtlebot_le2i remap_rplidar_minimal.launch (Bring up turtlebot)
```

We are going to configure to subscribe to combined odometry data (encoders + gyro) rather than **/odom** topic which only shows the encoder data. Hence we run this script

```
$roslaunch rbx1_bringup odom_ekf.launch
```

On Workstation:

run **rviz** to see the combined odometry data. This is displaying the combined odometry data on the **/odom_ekf** topic rather than just the wheel encoder data published on the **/odom** topic.

```
$rosrun rviz rviz -d 'rospack find rbx1_nav'/nav_ekf.rviz
```

finally we run the python script to do the previously defined task

```
$rosrun rbx1_nav timed_out_and_back.py
```

Now we can see from rviz is that, the robot didn't do actually what is supposed to do because of environmental constraint. It messed up but we have unfairly handicapped it by not using the odometry data in our script. Our script based only on time and speed. While the odometry data will not match the real motion exactly, it should give us a better result if we use it. We shall see it in the next step

1.6   Goal 2 : Advanced Motion of Mobile Base

Let's assume that we have some data from the laser in the form of distances from the laser's center point. In other words, we have some data in the **"base_laser"** coordinate frame. Now suppose we want to take this data and use it to help the mobile base avoid obstacles in the world. To do this successfully, we need a way of transforming the laser scan we've received from the **"base_laser"** frame to the **"base_link"** frame. In essence, we need to define a relationship between the **"base_laser"** and **"base_link"** coordinate frames.
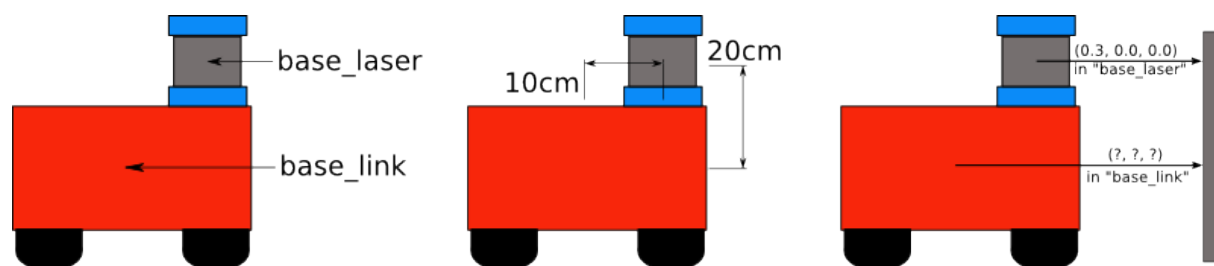
Conversion between coordinate frames

Such is the case for the turtlebot Odometry data we use to move the robot. While the **/base_link** frame corresponds to a real physical part of Controlling a Mobile Base robot, the **/odom** frame is defined by the translations and rotations encapsulated in the odometry data. These transformations move the robot relative to the **/odom**

7

frame. In ROS if we see the **nav_msgs/Odometry** message structure, **/odom** is used as the parent frame and **/base_footprint** (for turtlebot) as the **child_frame_id**. The transformation between these two frames, is with the help of tf library.

Rather than guessing distances and angles based on time and speed, our next script will monitor the robot's position and orientation as reported by the transformation between the **/odom** and **/base_footprint** frames.

Algorithm 2: ODOMETRY based out and back

```
rate = 50
goal_distance = 1.0 m
linear_speed = 0.2m/s
linear_duration = goal_distance / linear_speed;

angular_speed = 1.0
goal_angle = pi
angular_duration = goal_angle / angular_speed;

A: Repeat step 1 to 6 twice

  1. get initial position
  2. while distance < goal_distance
     robot will go at a speed set by linear_speed
     sleep 1 cycle
     get new linear position from ODOMETRY
     calculate new distance from new position and old position
  3. stop the robot
     Set last_angle = rotation
       turn_angle = 0
  4. while (turn_angle < goal_angle)
     start rotating
     sleep 1 cycle
     get new rotation position from ODOMETRY
     delta_angle = (rotation - last_angle)
     turn_angle += delta_angle
     last_angle = rotation
  5. stop the robot 1 cycle before next run

B: Program Terminates
```

We ssh into the turtlebot and run the follwing scripts on Turtlebot:

```
$roslaunch turtlebot_le2i remap_rplidar_minimal.launch (Bring up turtlebot)
```

We are going to configure to subscribe to combined odometry data (encoders + gyro) rather than **/odom** topic which only shows the encoder data. Hence we run this script

```
$roslaunch rbx1_bringup odom_ekf.launch
```

On Workstation: run rviz to see the combined odometry data This is displaying the combined odometry data on the **/odom_ekf** topic rather than just the wheel encoder data published on the **/odom** topic.

```
$rosrun rviz rviz -d 'rospack find rbx1_nav'/nav_ekf.rviz
```

finally we run the python script to do the previously defined task

```
$rosrun rbx1_nav odom_out_and_back.py
```

Using odometry the results improves significantly from the timed out-and-back case, because still accuracy and reliability of this process depends on the robot's internal sensors, the accuracy of the calibration procedure, and environmental conditions. Since the robot is performing a very simple task, the navigation error is slight, and it is more evident after performing the task successively several times.
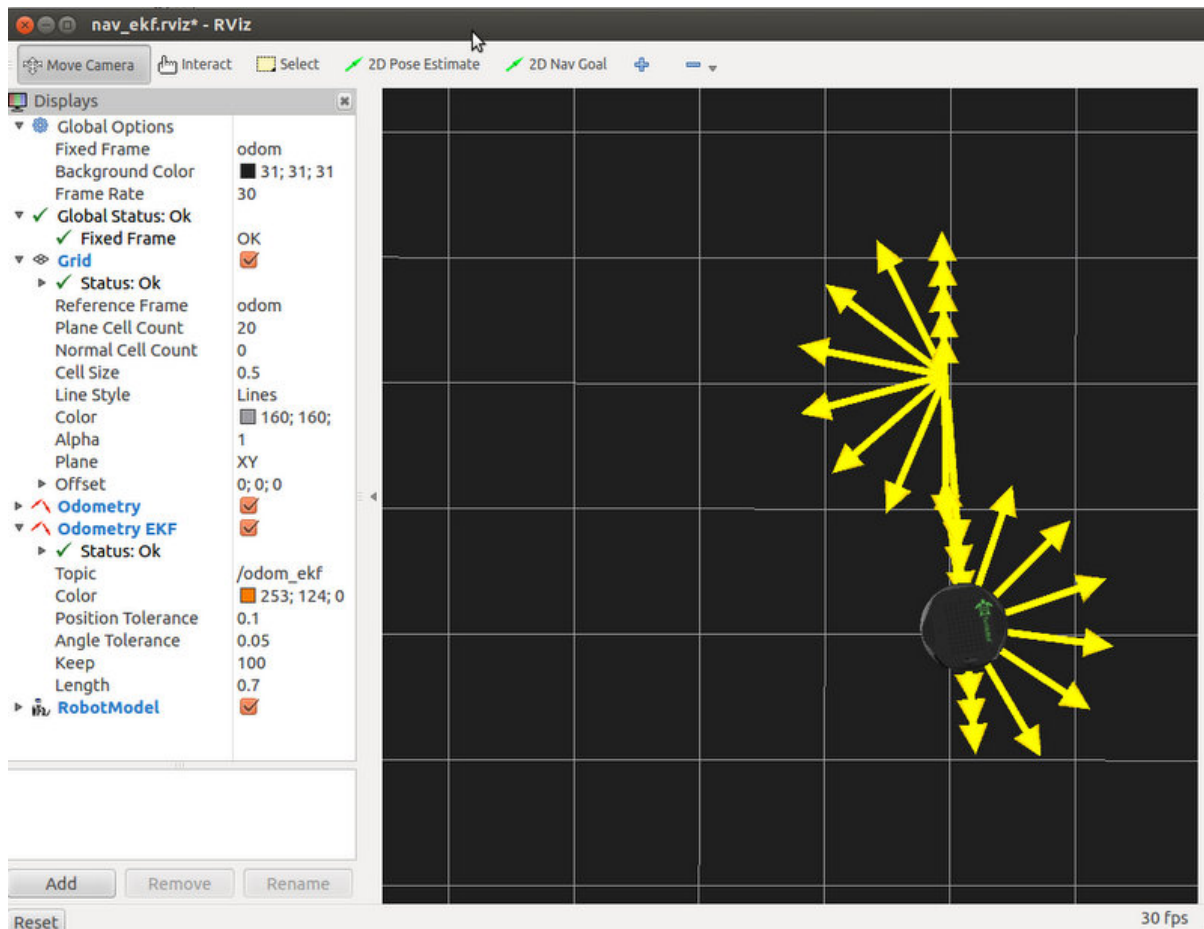
Odometry based out and back on Turtlebot

1.6.1 Dead Reckoning

Even small amount of error in the robot odometry data accumulates over time. Hence a robot navigating only using internal motion data and without any reference to external landmarks will grow on it's mistakes and eventually be completely lost. This is knows as Dead Reckoning.

We can improve this for our turtlebot using map and SLAM.

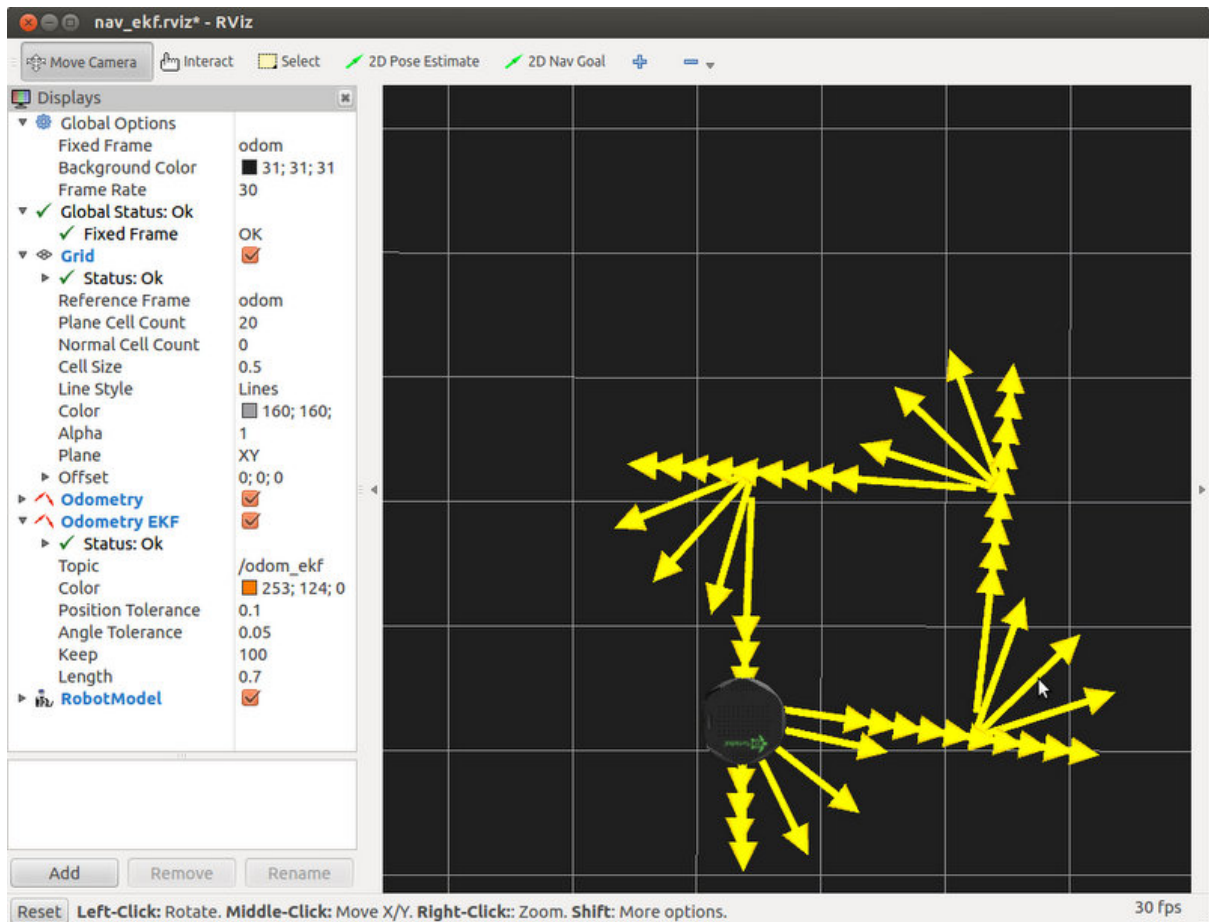## 1.7 Goal 3 : Navigating a Square using Twist+Odometry

In a similar way to the previous Goal, in Goal 3 the robot navigates through a square path, using both Twist and Odometry messages.

There are the commands we use to navigate using nav_square.py script

```
Turtlebot Bringup:
$roslaunch turtlebot_le2i remap_rplidar_minimal.launch (Bring up turtlebot)
Combined odometry data
$roslaunch rbx1_bringup odom_ekf.launch
Visualization
$rosrun rviz rviz -d 'rospack find rbx1_nav'/nav.rviz
Move in a square:
$rosrun rbx1_nav nav_square.py
```

The nav_square.py algorithm is nearly identical to the odometry-based out-and-back algorithm. The only difference is that it we now send the robot along four 1 meter legs using 90 degree rotations instead of two 1 meter legs using 180 rotations.

At the end of the run, we can see that the errors accumulated in odometry are visible, it didn't move in a perfect square:

Navigating a square using Twist+Odometry

1.8 Goal 4 : Navigation with Path Planning move_base

ROS provides the move_base package that allows us to specify a target position and orientation of the robot with respect to some frame of reference.

To use move_base package, it requires a map of the environment and the control points where the robot moves to. A launch file is created to load a map into a map_server node. In our case we used a blank map. A node is created to append control points into the blank map.

Following figure summarizes how the move_base path planner works. On the bottom left of figure is a base controller node for low level motion control. on the top right is the map server which provides a map of environment.
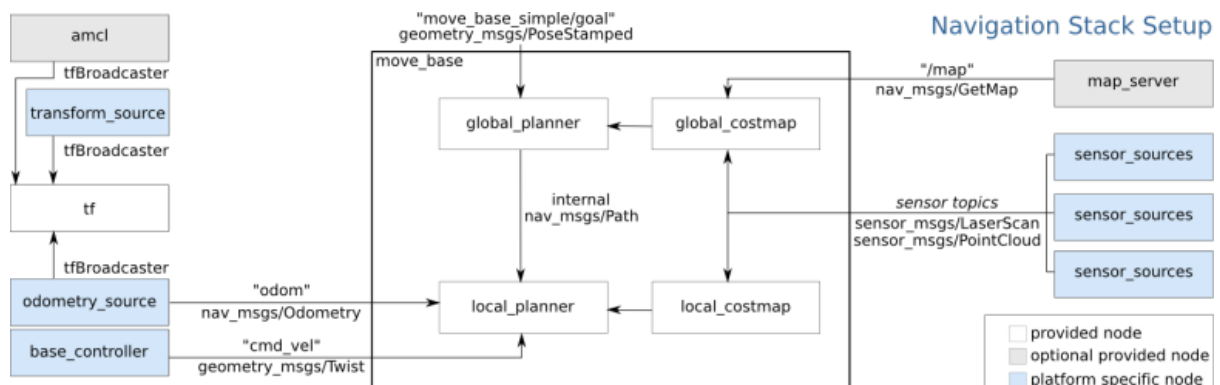
A general scheme of move_base path planner

There are the commands we use to navigate using move_base

```
Turtlebot Bringup:
$roslaunch turtlebot_le2i remap_rplidar_minimal.launch (Bring up turtlebot)
Combined odometry data
$roslaunch rbx1_bringup odom_ekf.launch
Launch planner (move_base) with a blank map
$roslaunch rbx1_nav tb_move_base_blank_map.launch
Visualization
$rosrun rviz rviz -d `rospack find rbx1_nav`/nav.rviz
Move in a square trajectory according to waypoints:
$rosrun rbx1_nav move_base_square.py
```
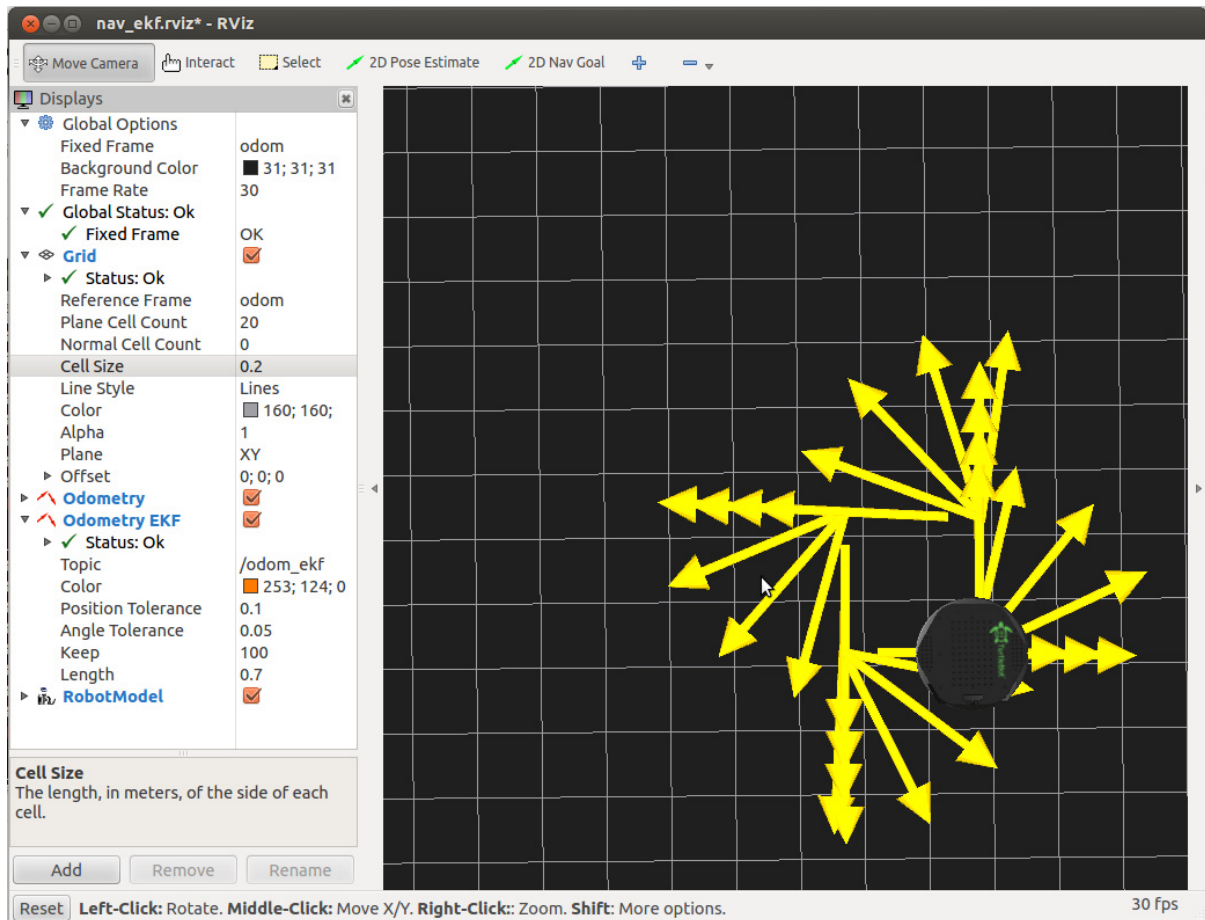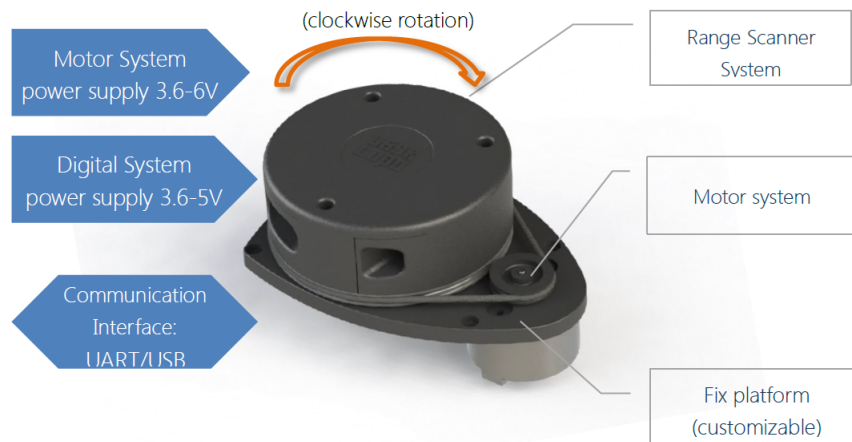

Moving using four waypoints in a square using move_base

A video of this movement can be seen here:

```
https://www.youtube.com/watch?v=5e61iTaRLl0
```

2 Part 2 - Planar Laser RangeFinder

2.1 RP-LIDAR

The different RP-LIDAR's parts

RP-LIDAR is a low-cost **360 degree** 2D Laser Scanner (LIDAR) system.
It can perform scanning with a frequency of **5.5Hz** when sampling 360 points within **6 meter range**.

RP-LIDAR is based on **laser triangulation ranging principle** :
It emits modulated infrared laser signal and the laser signal is then reflected by the object to be detected. The returning signal is sampled by vision acquisition system in RP-LIDAR and the DSP embedded in RP-LIDAR start processing the sample data, output distance value and angle value between object and RP-LIDAR through communication interface.
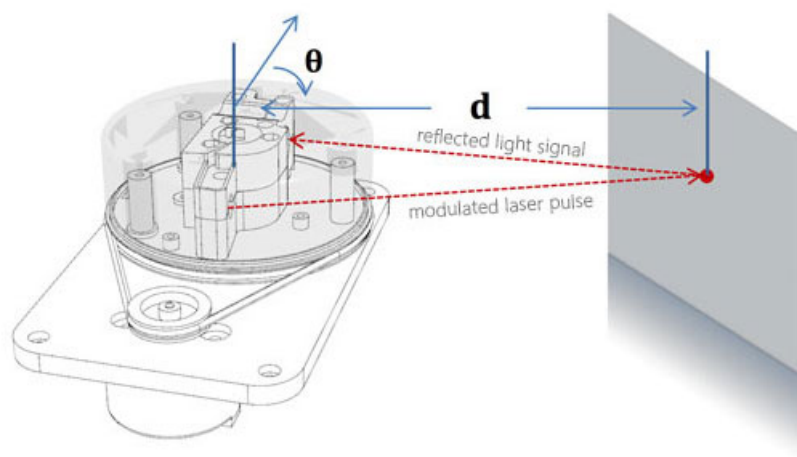

Graphical representation of laser triangulation

The system can measure distance data in more than 2000 times' per second and with high resolution distance output ($<1\%$ of the distance) :

Data Type	Unit	Description
Distance	mm	Current measured distance value
Heading	degree	Current heading angle of the measurement
Quality	level	Quality of the measurement
Start Flag	(Boolean)	Flag of a new scan

Output data given by RP-LIDAR

These data will be used in order to scan a map and to move the turtlebot in function of this map and the points detected.
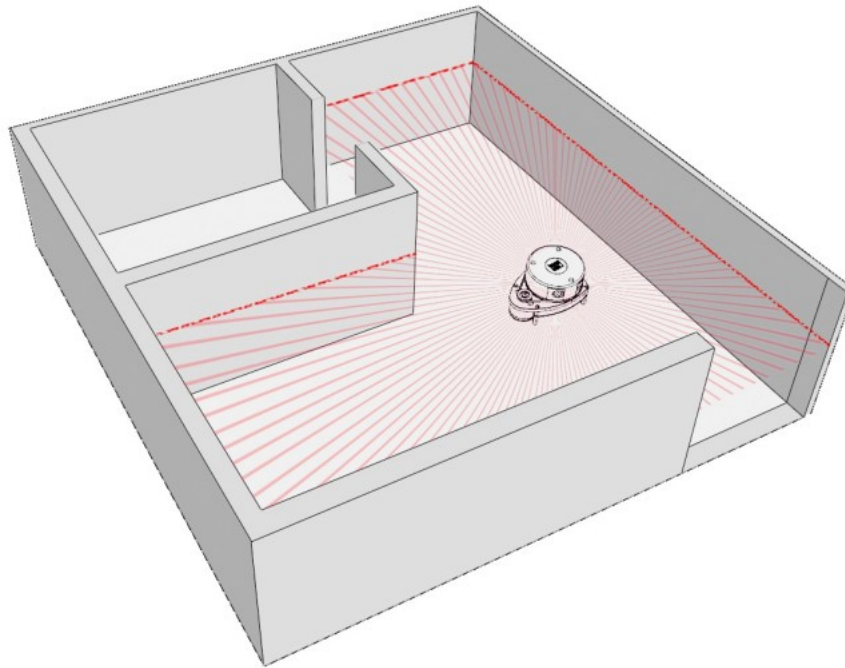
Example of scanned map using RP-LIDAR

2.2 Integration on Turtlebot2

2.2.1 rplidar_ros package

The package **"rplidar_ros"** needs to be added to integrate the RP-LIDAR to ROS. As this package is only available in **Ros Hydro**, this ROS distribution will be used (the buildsystem **"Catkin"** is also required).
Once it has been done, the most important informations we want are located in the rplidar node (**rplidarNode**).
The **rplidarNode** is a driver for RP-LIDAR which reads RP-LIDAR raw scan result using RPLIDAR's SDK and convert the informations to messages in the topic **sensor_msgs/LaserScan**.

This node is created with the help of the **rplidar.launch** file, located in the **"rplidar_ros"** package. All the parameters needed for the Lidar are stated in this file :

```
<launch>
 <node name="rplidarNode"  pkg="rplidar_ros" type="rplidarNode" output="screen">
 <param name="serial_port" type="string" value="/dev/ttyUSB1"/>
 <param name="serial_baudrate" type="int" value="115200"/>
 <param name="frame_id"    type="string" value="base_laser_link"/>
 <param name="inverted"    type="bool" value="false"/>
 <param name="angle_compensate" type="bool" value="true"/>
 </node>
</launch>
```

2.2.2 turtlebot_le2i package

Then it is necessary to create a new package where the position of the Lidar on the turtlebot, compared with the "[parent]" is given (which in our case is the **plate_top_link**). This will be used by the **tf** node, to compute the laser position compared to the base.
In our case, this package is named **"turtlebot_le2i"**. It contains a bringup launch file called **rplidar_minimal.launch** which launch the simple bringup **minimal.launch**, the **rplidar.launch** and also some .xml files.
In these files, one is the **rplidar.urdf.xacro**. It contains all about the RP-LIDAR position : the origin, the box size, ... Here is a part of this file :

```
<xacro:macro name="sensor_rplidar" params="parent">
<joint name="laser" type="fixed">
  <origin xyz="0.180 0.00 0.040" rpy="0 3.14159265 0" />
  <parent link="${parent}" />
  <child link="base_laser_link" />
</joint>

<link name="base_laser_link">
  <visual>
    <geometry>
      <box size="0.00 0.05 0.06" />
    </geometry>
    <material name="Green" />
  </visual>
  <inertial>
    <mass value="0.000001" />
    <origin xyz="0 0 0" />
    <inertia ixx="0.0001" ixy="0.0" ixz="0.0"
    iyy="0.0001" iyz="0.0" izz="0.0001" />
  </inertial>
</link>
</xacro:macro>
```
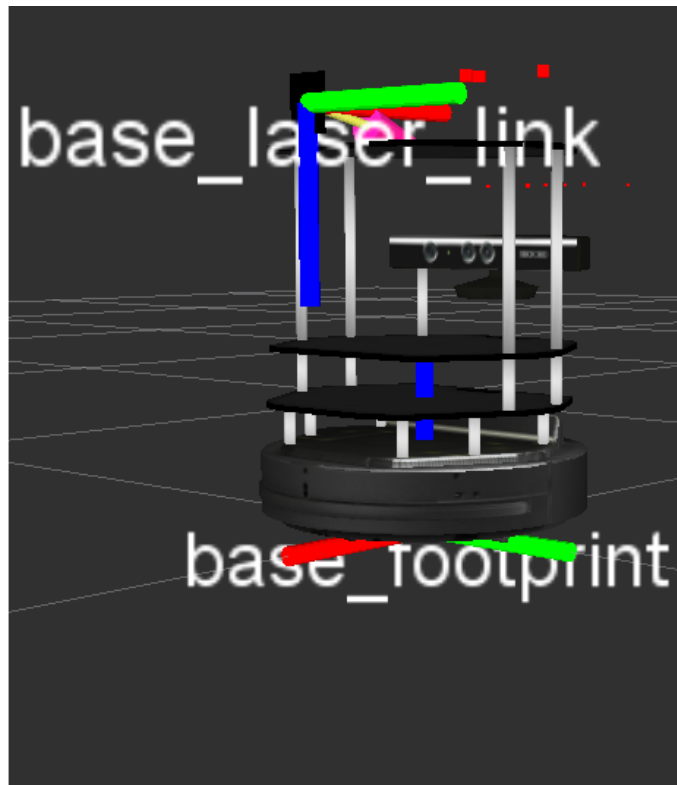
Note : Xacro is an XML macro language, used to construct shorter and more readable XML files.

2.2.3 base_laser_link

You can see in the **"Relative Position"**, the values are the same as the coordinates stated in the .urdf (cf. *Configure Ros*).

There is also the computed **"Position"** from the base to the lidar, thanks to the **tf** node taking into account the relative position from the **plate_top_link**, which has also a relative position compared with his parent and so on until the**base_footprint**.

2.2.4 Displaying the data

To extract the data from the RP-LIDAR :
bringup the turtlebot,
then start a rplidar node and run **rplidar client process** to print the raw scan result :

```
$ roslaunch rplidar_ros rplidar.launch
$ rosrun rplidar_ros rplidarNodeClient
```

To see the data coming from the RP-LIDAR :
bringup the turtlebot,
then start a rplidar node and view the scan result in rviz:

```
$ roslaunch rplidar_ros view_rplidar.launch
```
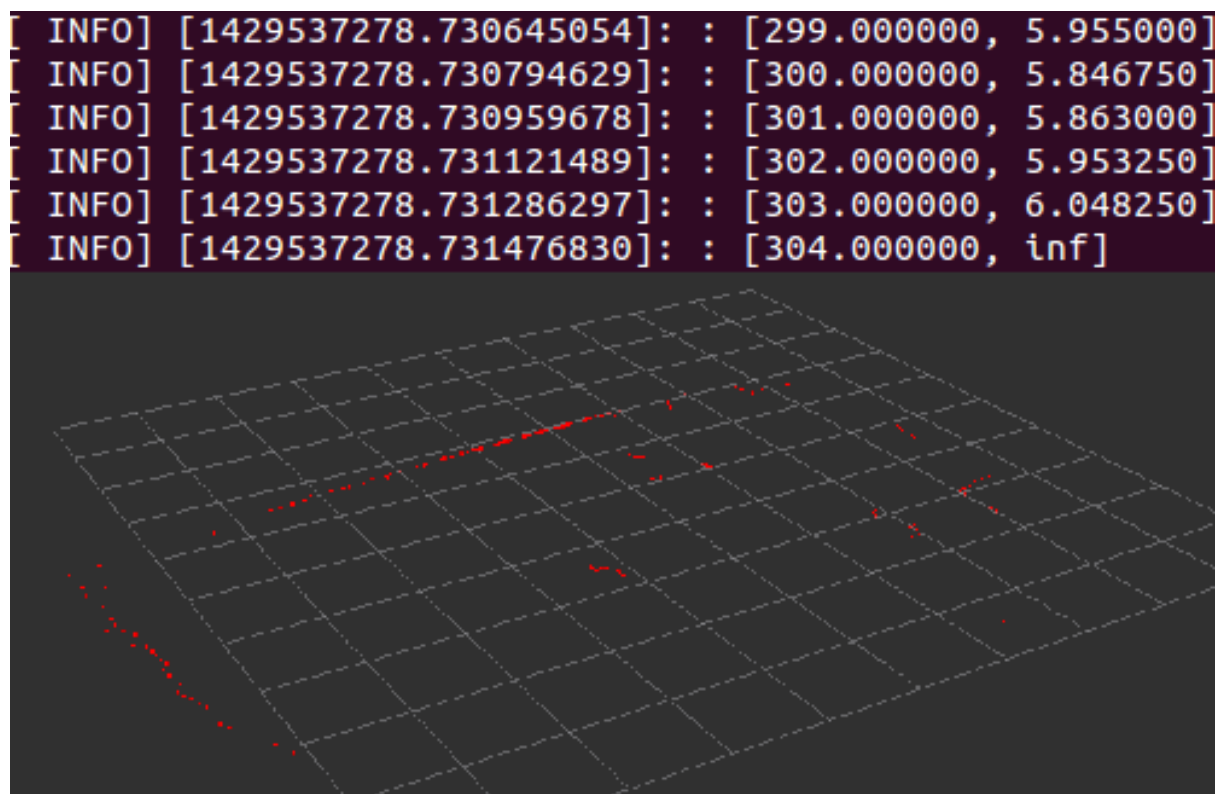

This scanner will be used here, for the application of general simultaneous localization and mapping (SLAM).

3 Part 3 - Navigation & Localization

Three essential ROS packages that make up the core of navigation stack:
- gmapping for creating map from RPlidar laser scan data
- amcl for the localization using an existing map
- move_base for moving the robot to a goal within given reference frame
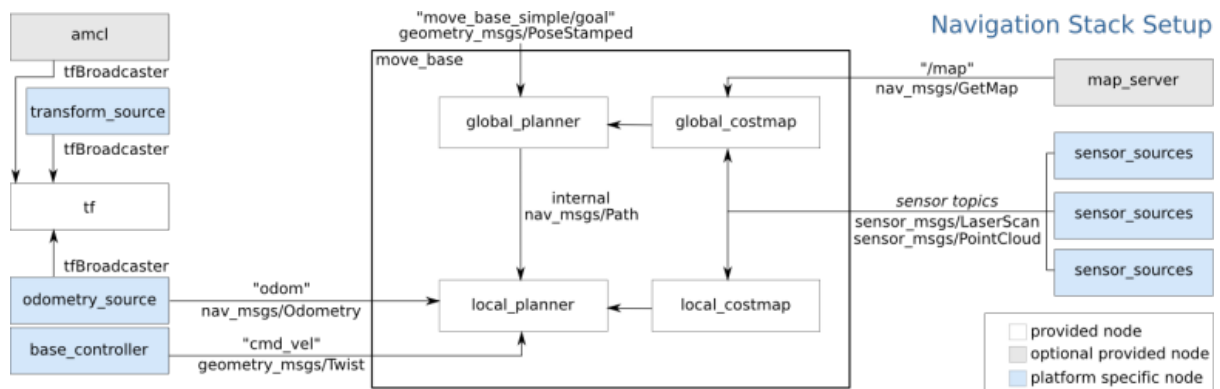
3.1   Map Creation

Although ROS based ROBOT's can navigate in an unknown environment, it's very ideal to create and use a map. In a known environment path planning/movement is much easier. In the project, gmapping package was used to create a map with the help of teleop.

3.1.1   Mapping

Learning maps is one of the fundamental problems in mobile robotics. Maps allow robots to efficiently carry out their tasks, such as localization, path planning, activity planning, etc. There are different ways to represent the world space, such as: Grid maps, Geometric maps, Voronoi graphs

Occupancy Grid Map, Maps the environment as a grid of cells. Cell sizes typically range from 5 to 50 cm, each cell holds a probability value that the cell is occupied in the range [0,100], Unknown is indicated by -1, Usually unknown areas are areas that the robot sensors cannot detect (beyond obstacles)

White pixels represent free cells
Black pixels represent occupied cells
Gray pixels are in unknown state

3.1.2   Marking and Clearing

The grid map is built using a process called **marking** and clearing. A marking operation inserts obstacle information into the map. A **clearing** operation removes obstacle information from the map. It consists of **raytracing** through a grid from the origin of the sensor outwards for each observation reported.

3.1.3   SLAM

Simultaneous localization and mapping (**SLAM**) is a technique used by robots to build up a map within an unknown environment while at the same time keeping track of their current location.

3.1.4 Gmapping

Gmapping (http://wiki.ros.org/gmapping) package provides laser-based SLAM as a ROS node called **slam_gmapping**. It Uses the FastSLAM algorithm. It takes the laser scans and the odometry and builds a 2D occupancy grid map. It updates the map state while the robot moves.

3.2 SLAM Map Building with TurtleBot

In order to navigate and localization the turtlebot, first the map of the arena needs to be created. Turtlebot already got the essential updates in order to use gmapping package while it been used for the navigation purpose describe earlier.

Following steps been used to create and save the map of arena:
To launch the gmapping package
on the TurtleBot :

```
$ roslaunch turtlebot_le2i rplidar_minimal.launch
$ roslaunch rbx1_nav rplidar_gmapping_demo.launch
```

For launching rviz in order to view and save the map
on the Workstation :

```
$ roslaunch turtlebot_rviz_launchers view_navigation.launch
$ roslaunch turtlebot_teleop keyboard_teleop.launch
```
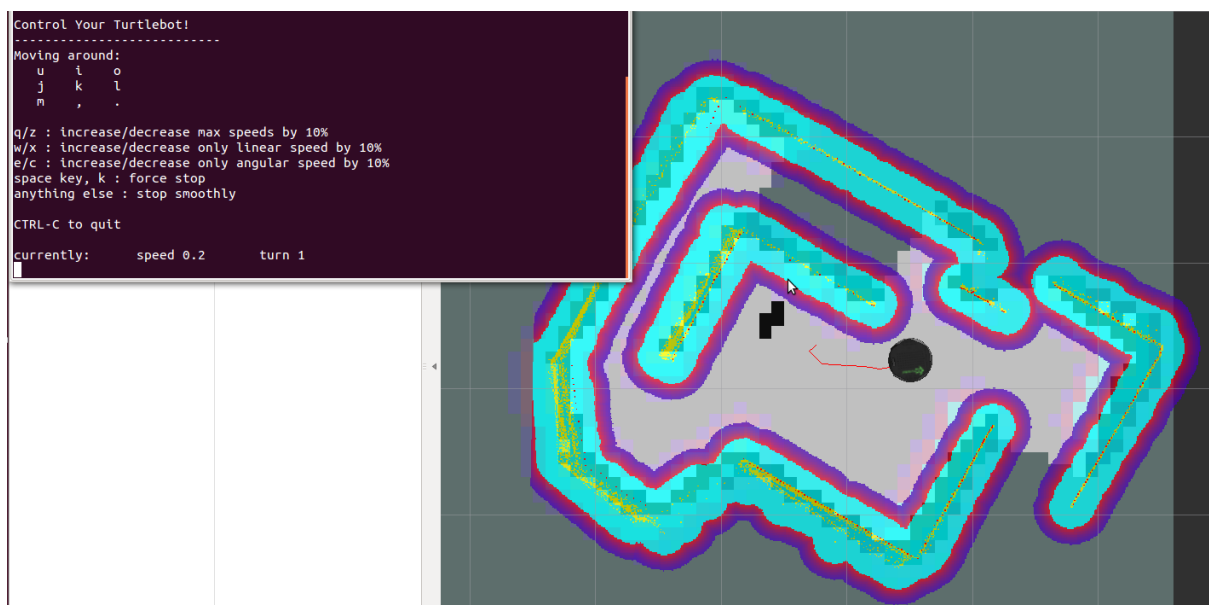

3.2.1 Saving Map

The robot needed to drive around using telop keyboard slowly to get the entire arena. This process is used few times to circle the arena in order to record better map. **map_server** package is used by ROS to load and save maps.
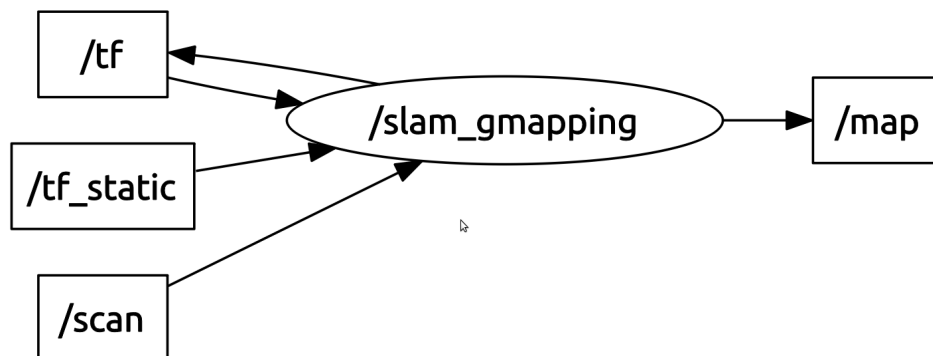
- Save the map to file:

```
$ rosrun map_server map_saver -f /tmp/my_map
```

map_saver generates the following files in the directory:
my_map.pgm – the map itself
my_map.yaml – the map's metadata
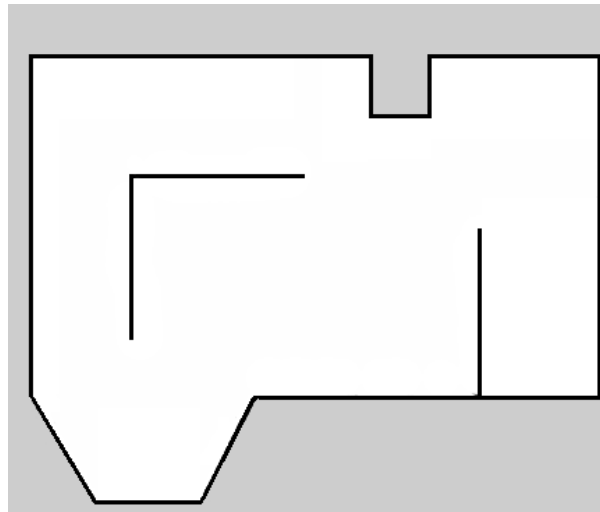
my_map.pgm

Some important fields:
- **resolution:** Resolution of the map, meters / pixel
- **origin:** The 2-D pose of the lower-left pixel in the map as (x, y, yaw)
- **occupied_thresh:** Pixels with occupancy probability greater than this threshold are considered completely occupied.
- **free_thresh:** Pixels with occupancy probability less than this threshold are considered completely free.

3.3 Localization

Localization is the problem of estimating the pose of the robot relative to a map. Localization is not terribly sensitive to the exact placement of objects so it can handle small changes to the locations of objects. ROS uses the amcl package for localization. Before using the navigation, it's important to understand how amcl pacakage works.

3.4 Amcl

amcl is a probabilistic localization system for a robot moving in 2D. It implements the adaptive (or KLD-sampling) Monte Carlo localization approach, which uses a particle filter to track the pose of a robot against a known map. amcl takes in a laser-based map, laser scans, and transform messages, and outputs pose estimates. On startup, amcl initializes its particle filter according to the parameters provided. As defaults, if no parameters are set, the initial filter state will be a moderately sized particle cloud centered about (0,0,0).
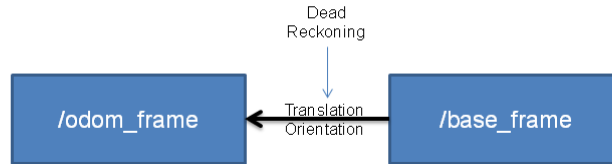
Subscribed topics:
- **scan** – Laser scans
- **tf** – Transforms
- **initialpose** – Mean and covariance with which to (re-) initialize the particle filter
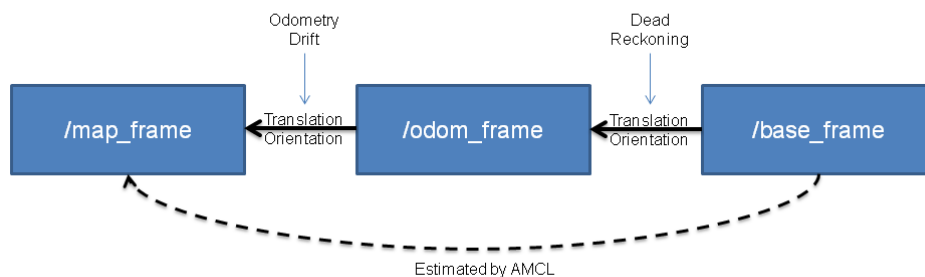- **map** – the map used for laser-based localization

Published topics:
- **amcl_pose** – Robot's estimated pose in the map, with covariance
- **Particlecloud** – The set of pose estimates being maintained by the filter

Odometry Localization

Dead Reckoning

/odom_frame ← Translation Orientation /base_frame

AMCL Map Localization

Odometry Drift

Dead Reckoning

/map_frame ← Translation Orientation /odom_frame ← Translation Orientation /base_frame

Estimated by AMCL

The difference between odometry based localization and AMCL map localization

3.5   Navigation and localization using a map and amcl

Navigating a robot in a map is done simply using amcl, existing map and Rviz. To run the navigation stack with amcl, the following
- **map_server** – for loading the map
- **amcl** – for the localization system
- **move_base** – manages the navigation stack

Following steps, been used to run navigation
on the turtlebot netbook:

```
$ roslaunch turtlebot_le2i remap_rplidar_minimal.launch
$ roslaunch rbx1_nav tb_demo_amcl.launch map:=my_map.yaml
```

on the workstation :

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

3.5.1   Point and click navigation using Rviz

After amcl bring up, the initial pose of the turtlebot is to be set, because amcl can't do its own. To set the initial
pose, just need to click on the "2D Pose Estimate" button in rviz. Then click on the point in the map where the
robot is located. While holding down the mouse button, a large green arrow appears. The mouse then moved to
orient the arrow to match the orientation of the robot, and then released. With the robots initial position set, it's
now possible to use the "2D Nav Goal" button in rviz to navigate goals of the turtlebot at different locations in the
map of arena.

3.5.2 Path Planning

Path planning is based on WHERE am I and HOW to go there technique. To move from position A to position B turtlebot needs to know its initial position and path finding algorithm to reach its destination. '2D pose estimate' uses the amcl package to set turtlebot position in the map. Path planning uses **Global plan** and **Local plan** to reach the goal. To understand the navigation it is very important to understand how both plan combined in order to reach the goal.

At the first try, the turtlebot was not able to reach its destination due to a L shape obstacle in the middle, the turtlebot continuously stacked and collide into the edge of the wall. After investigation, it is discovered that robots inflation is the factor in order to run the turtlebot smoothly in the arena. In order to fix the issues, it is very important to go through all the files contained by move_base package. Using the values of local cost map, global cost map and the robot footprint, 2D_Nav_Goal determines the path and distance in order to reach the goal.

Example of collisions

3.5.3 Move_base package

The move_base package provides an implementation of an action that, given a goal in the world, will attempt to reach it with a mobile base. The move_base node links together a global and local planner to accomplish its global navigation task. It supports any global planner adhering to the **nav_core::BaseGlobalPlanner** interface specified in the nav_core package and any local planner adhering to the **nav_core::BaseLocalPlanner** interface specified in the nav_corepackage. The move_base node also maintains two costmaps, one for the global planner, and one for a local planner that are used to accomplish navigation tasks.

Move_base nodes requires four configuration files before it can run. These files defines a number of parameters related to the cost of running into obstacle, the radius of the robot, how fast the robot moves and how far the path planner should looks.

The four configuration files can be found in the **config** subdirectory of the **rbx1_nav** package and are called:

- base_local_planner_params.yaml (see Appendix 1)
- costmap_common_params.yaml (see Appendix 2)
- global_costmap_params.yaml (see Appendix 3)
- local_costmap_params.yaml (see Appendix 4)

3.5.4 Cost Map

Cost map is a data structure that represents places that are safe for the robot to be in a grid of cell. It is based on the occupancy grid map of the environment and user specified inflation radius. There are two types of costmaps in ROS: Global costmap is used for global navigation and Local costmap is used for local navigation.

3.5.5 Cost Map Value :

Each cell in the costmap has an integer value in the range [0, 255].There are 5 special symbols for cost values:
　　- **NO_INFORMATION** (255) - Reserved for cells where not enough information is sufficiently known

- **LETHAL_OBSTACLE** (254) - Indicates a collision causing obstacle was sensed in this cell
- **INSCRIBED_INFLATED_OBSTACLE** (253) - Indicates no obstacle, but moving the center of the robot to this location will result in a collision
- **POSSIBLY CIRCUMSCRIBED** (128-252) – If the robot center lies in that location, then it depends on the orientation of the robot whether it collides with an obstacle or not
- **FREE_SPACE** (0) - Cells where there are no obstacles and the moving the center of the robot to this position will not result in a collision
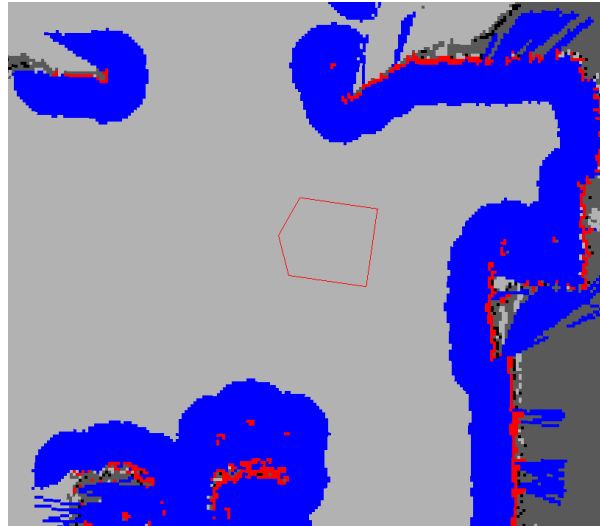
Example of cost map ros.wiki

3.5.6 Inflation Function :

Inflation is the process of propagating cost values out from occupied cells that decrease with distance. A brief explanation of inflation is given below:
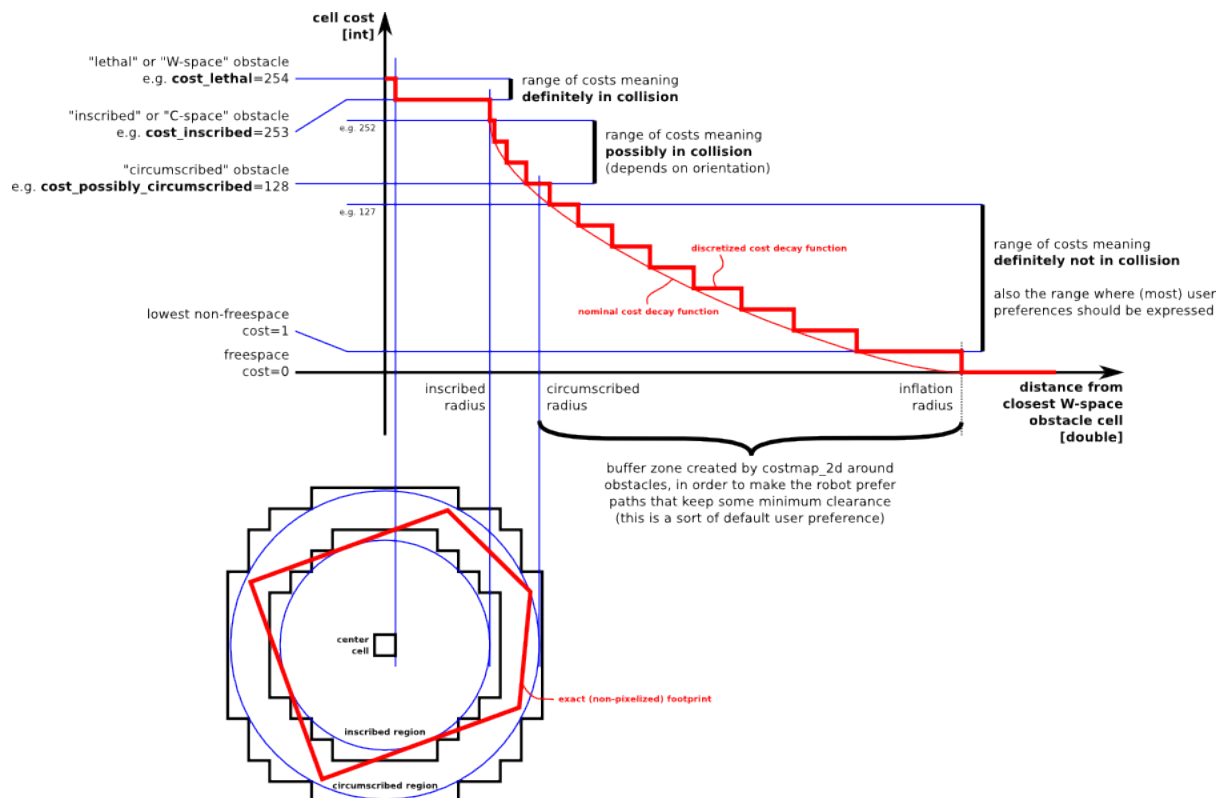
3.5.7 Global Planner

Global planner is based on static global cost map, start point and end point. As discussed earlier the cost map, depending on the cost, a map called global cost map calculated in order to create a global plan.

Map + Costs = Global Cost Map
Global Cost Map + Start + Target = Global Plan (way points)

3.5.8 Local Planner

Global map is static and using only global map can create problems especially when dynamic objects or obstacles are placed in the map. For the local path planner cost is calculated as below: Cost = path_distance_bias * (distance to path from the endpoint of the trajectory in map cells or meters depending on the meter_scoring parameter) + goal_distance_bias * (distance to local goal from the endpoint of the trajectory in map cells or meters depending on the meter_scoring parameter) + occdist_scale * (maximum obstacle cost along the trajectory in obstacle cost (0-254)).

3.6 Navigation in action :

The turtlebot navigation is tested using different initial pose and different goal pose. Following video shows the turtlebot in the arena moving from one side to another using 2D_nav_goal.
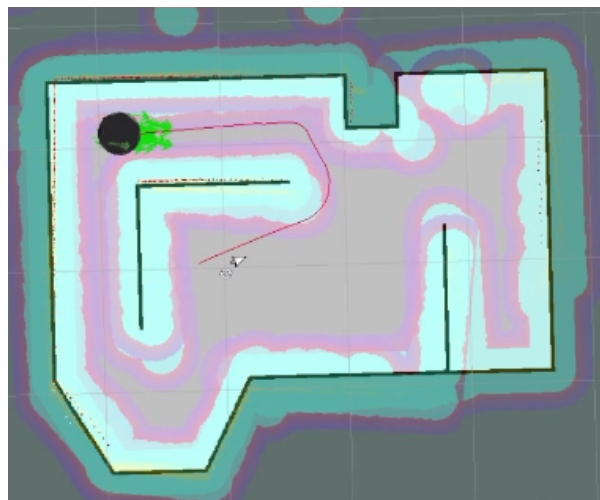
Example of navigation

Here is the result of the navigation in video:

```
https://www.youtube.com/watch?v=uNPqKQI8f8o
```

4 Part 4 - Controlling with Android and Others

4.1 Testing a node - Auto Docking

We launch the kobuki core by typing:

```
$ roslaunch kobuki_auto_docking compact.launch --screen
```

But it is just loading the algorithm, which is still not active. The algorithm is implemented as a typical action server. We need to call the server via an action client:

```
$ roslaunch kobuki_auto_docking activate.launch --screen
```

For our case it did find the dock perfectly but didn't set correctly due to hardware errors.
A video of the Auto Docking can be seen in this link:

```
https://www.youtube.com/watch?v=fQKv0jhXQxY
```

4.2   Testing a node - Turtlebot Follower

We initiate the turtlebot from our workstation:

```
$ ssh turtlebot@192.168.0.100 , Password:napelturbot
$ roslaunch turtlebot_le2i remap_rplidar_minimal.launch (Bring up turtlebot)
$ roslaunch turtlebot_follower follower.launch (Start follower demo)
```

Now the Turtlebot detects and follows the person/obstacle in front of it. [link of the video]

4.3   Android Teleoperation

4.3.1   Android Code Explanation

We were able to control our turtlebot using Android Teleop application which also gives a video feedback. We installed necessary Android development environment in our workstations and ran basic Android application. Following is the code and explanation of how the Android Teleop works:

```java
public class MainActivity extends RosActivity {

    private int cameraId = 0;
    private RosCameraPreviewView rosCameraPreviewView;
    private Handler handy = new Handler();

    public MainActivity() {
        super("CameraTutorial", "CameraTutorial");
    }
```

The MainActivity extends RosActivity. All ROS Android apps should extend this class to be compatible with the **android_remocons interface**. Extending this class gives some convenient ROS functionality/compatibility for free. RosActivity deals with initialising nodes, etc in Android.

We also make a **cameraId** in case there are multiple cameras, so we can keep track and switch between them.

We need a view, which is a **RosCameraPreviewView**. This view will display the images from the device's camera(s) on the screen. We imported that from an external library above. Our constructor can just be the default and call to the parent constructor.

```java
@Override
  protected void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      requestWindowFeature(Window.FEATURE_NO_TITLE);
      getWindow().addFlags(WindowManager.LayoutParams.FLAG_FULLSCREEN);
      setContentView(R.layout.activity_main);
      rosCameraPreviewView = (RosCameraPreviewView)
          findViewById(R.id.ros_camera_preview_view);
  }
```

Then we fill in the OnCreate() function, which is called when the app is opened. We may want to retrieve information from the last time the app was opened, which is where the savedInstanceState comes in. This is also where we can create views and set up the app's appearance. Later we'll configure this more from the activity_main.xml that defines the app layout. We can retrieve information from that file. Right now it just contains default information, but we'll change it later.

```
@Override
  protected void init(NodeMainExecutor nodeMainExecutor) {
      NodeConfiguration nodeConfiguration =
          NodeConfiguration.newPublic(InetAddressFactory
          .newNonLoopback().getHostAddress());
      nodeConfiguration.setMasterUri(getMasterUri());
      nodeMainExecutor.execute(rosCameraPreviewView, nodeConfiguration);
      handy.post(sizeCheckRunnable);
  }
```

This init() function comes from extending the RosActivity which helps us to handle the configuration of our ROS node(s) and other administration. We want to make sure our initial cameraId is zero. Then we connect up our view to a camera. So the app will start out showing and transmitting images from the "first" camera. We configure our connection to the "robot", netbook computer. When the app is started, we give it the ROS_MASTER_URI. The app uses this to connect to the netbook computer and the ROS master running on it. Once it has that information it can configure its node(s) and send images to the computer. Once the nodes have been initialized, we will setup the preview for the camera. We dont want to set the camera before the view is ready, so we run a repeating task to check the height and width. Once those have been set, we set the camera as well.

```
@Override
  public boolean onTouchEvent(MotionEvent event) {
      if (event.getAction() == MotionEvent.ACTION_UP) {
          //....
              rosCameraPreviewView.setCamera(Camera.open(cameraId));
          //....
      return true;
  }
}
```

Then On Touch Event we set the rosCameraPreviewView to show images from a camera with the camera ID and show user a Toast message.

In the AndroidManifest.xml file we define our package name and version number and also the minimum Android SDK requirements. Also we give the app various access to, for example access to the camera, the state of the wireless connection, etc. We also define the app icon, orientation and display name. Using intent-filters we decide how activities in our app can be accessed.

There is a file generated in MyApplication/src/main/res/layout called activity_main.xml. This file helps define how our app will look.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
          android:layout_width="fill_parent"
          android:layout_height="fill_parent"
          android:orientation="vertical" >

  <org.ros.android.view.camera.RosCameraPreviewView
          android:id="@+id/ros_camera_preview_view"
          android:layout_width="fill_parent"
          android:layout_height="fill_parent" />

</LinearLayout>
```

4.3.2 Preparing the Device

You need to be **directly** connected to the robot with your Android Device. In our case, the network is named :

```
TURTLEBOT-E-5GHz
```

Note : This is important to have an Android Device supporting 5GHz Wi-Fi, otherwise you need to set up the router with a 2.4 GHz.

-Connect your device to this Wi-Fi.
-Download **"Ros Teleop (Hydro)"** in the Play Store.

ROS Teleop (Hydro)
OSRF

DÉSINSTALLER | OUVRIR
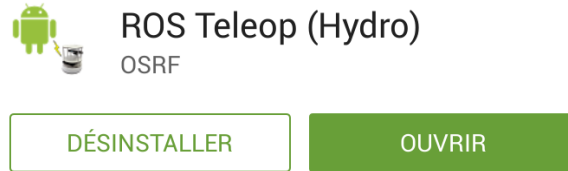
Now your device is ready.

4.3.3   Different bringup

There are three different ways to bringup the turtlebot :

-The **Minimal**, which we used previously.

And two others :

-The **App Manager**, which can do everything minimal does, but also offers the option of managing your programs as robot apps via the app manager.

-And the **Android Enabled**, which starts private/public masters and allows the app manager to be controlled by a remote android device via the public master.

4.3.4   Concert

What will be used for controlling with Android is the following statement :
on the turtlebot connected to the workstation with the same Wi-Fi as the Android Device :

```
$ rocon_launch turtlebot_bringup bringup.concert
```

The concert is a multimaster framework running on top of the interactions and **rocon_launch** is a multimaster version of roslaunch.

These ones allow us to launch the last two bringup launch files. This is exactly what contains the **bringup.concert** :

```
<concert>
   <launch package="turtlebot_bringup" name="paired_public.launch" port="11311"/>
   <launch package="turtlebot_bringup" name="minimal_with_appmanager.launch"
       port="11312"/>
</concert>
```

It launches two linux shells (gnome-terminal or konsole) which spawn the following ros master environments :

One is the **Private Master** :

-Port: 11312

-Software: This master runs just the rocon gateway and robot app_manager with a few default software, this means complex navigation as SLAM is not possible in this linux shell.

The other is the **Public Master** :

-Port: 11311

-Software: This master runs another rocon gateway and a simple pairing script that assists the invitation of the private app_manager and flipping the appropriate topics back and forth between the two masters. This one will be accessed by the android application.

In the application, you need to enter the public master, so you need to write the following URI to find the turtlebot :

```
http://192.168.0.100:11311/
```
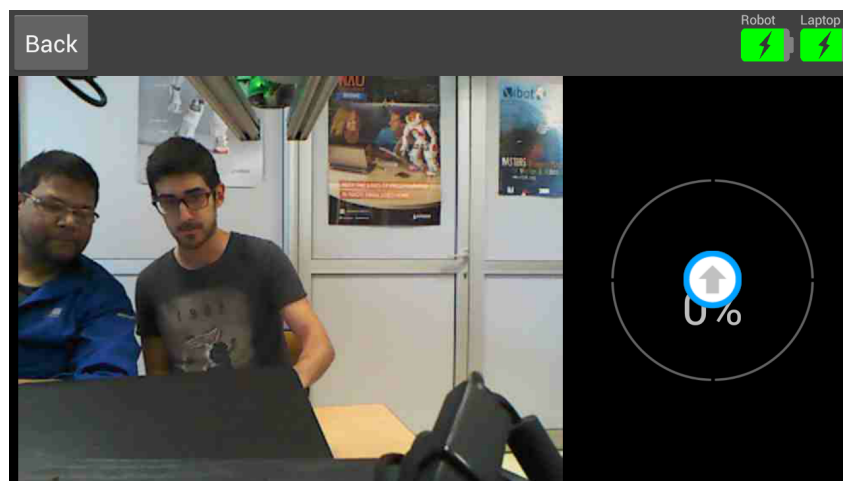

ScreenShot of the running application

Here is the video of the result :

```
https://www.youtube.com/watch?v=pBYmtod4o18
```

4.3.5 Possible extension

Multi-Master Concert: After successfully paired up with android device, an approach taken to create a multi-master concert launch files in order to use RVIZ in the PC. Goal was to make custom concert file to bring up the turtlebot with RPLIDAR minimum launch with android support. If the plan successfully implemented, while the turtlebot connected with android device, RVIZ could be run in the PC to view the navigation.

Although the custom concert fired up without any error, it failed to pair up with android device. Neither the RVIZ ran in the PC. For the limitation of the time it wasn't possible to investigate further. Custom concert files are attached in appendix 5.

5   Conclusion

Using RPLidar with the turtlebot is a very economical way to implement a cost effective autonomous navigation solution. Although RPLidar scan data is not highly accurate, it has been successfully implanted in the navigation process. The Android app that was used for the android part is no longer available for download from google play store. After a brief communication between the team and Yujin Robot, it is clear that OSRF is not interested in building or updating any android app for ROS Hydro. Indigo version of application by yujin robot is available to use. So for further android based development, it is highly recommended to use ROS indigo.

6 Appendix

6.1 Appendix 1

base_local_planner_params

```
controller_frequency: 3.0
recovery_behavior_enabled: false
clearing_rotation_allowed: false

TrajectoryPlannerROS:
  max_vel_x: 0.3
  min_vel_x: 0.05
  max_vel_y: 0.0 # zero for a differential drive robot
  min_vel_y: 0.0
  max_rotational_vel: 1.0
  min_in_place_vel_theta: 0.5
  escape_vel: -0.1
  acc_lim_x: 2.5
  acc_lim_y: 0.0 # zero for a differential drive robot
  acc_lim_th: 3.2

  holonomic_robot: false
  yaw_goal_tolerance: 0.3 # about 6 degrees
  xy_goal_tolerance: 0.2 # 10 cm
  latch_xy_goal_tolerance: false
  pdist_scale: 0.8 # changed
  gdist_scale: 1.0 # changed
  meter_scoring: true

  heading_lookahead: 0.325
  heading_scoring: false
  heading_scoring_timestep: 0.8
  occdist_scale: 0.2 # changed to low
  oscillation_reset_dist: 0.05
  publish_cost_grid_pc: false
  prune_plan: true

  sim_time: 1.0
  sim_granularity: 0.025
  angular_sim_granularity: 0.025
  vx_samples: 8
  vy_samples: 0 # zero for a differential drive robot
  vtheta_samples: 20
  dwa: true
  simple_attractor: false
```

6.2 Appendix 2

costmap_common_params

```
obstacle_range: 2.5
raytrace_range: 3.0
robot_radius: 0.22 # chngd
inflation_radius: 0.5 # chngd
max_obstacle_height: 0.6
min_obstacle_height: 0.0
observation_sources: scan
scan: {data_type: LaserScan, topic: /scan, marking: true, clearing: true,
    expected_update_rate: 0.0}
```

6.3 Appendix 3

global_costmap

```
global_frame: /map
robot_base_frame: /base_footprint
update_frequency: 5.0 # chng frm 3 to 5
publish_frequency: 0 #chngd to previous 0
static_map: true
rolling_window: false
resolution: 0.1
transform_tolerance: 1.0
map_type: costmap
```

6.4 Appendix 4

local_costmap

```
global_frame: /odom
robot_base_frame: /base_footprint
update_frequency: 3.0 # chngd frm 3.0
publish_frequency: 1.0 # chngd frm 2.0
static_map: false
rolling_window: true
width: 6.0
height: 6.0
resolution: 0.01 # chngd frm .01
transform_tolerance: 1.0
map_type: costmap
```

6.5 Appendix 5

Customized concert ("bringup_copy.concert") :

```xml
<concert>
    <launch package="turtlebot_bringup" name="paired_public.launch" port="11311"/>
    <launch package="turtlebot_bringup" name="minimal_with_appmanager_copy.launch"
        port="11312"/>
</concert>
```

"minimal_with_appmanager_copy.launch" :

```xml
<launch>
 <!-- Turtlebot -->
 <arg name="base" default="$(optenv TURTLEBOT_BASE kobuki)"/> <!-- create, rhoomba
     -->
 <arg name="battery" default="$(optenv TURTLEBOT_BATTERY
     /sys/class/power_supply/BAT0)"/> <!-- /proc/acpi/battery/BAT0 in 2.6 or
     earlier kernels-->
 <arg name="stacks" default="$(optenv TURTLEBOT_STACKS hexagons)"/> <!-- circles,
     hexagons -->
 <arg name="3d_sensor" default="$(optenv TURTLEBOT_3D_SENSOR kinect)"/> <!--
     kinect, asus_xtion_pro -->
 <arg name="simulation" default="$(optenv TURTLEBOT_SIMULATION true)"/>

 <!-- App manager -->
 <arg name="robot_name" default="$(optenv ROBOT_NAME turtlebot)"/>
 <arg name="robot_type" default="$(optenv ROBOT_TYPE turtlebot)"/>
 <!-- By default, we only tune into the pairing master's hub, so set the whitelist
     here for "Internal Hub" -->
 <!-- To run concert mode, set an empty string or use your concert's hub name
     specifically -->
 <!-- For concert mode, you should also disable the auto_invite in paired
     public.launch -->
 <arg name="hub_whitelist" default="$(optenv ROBOT_HUB_WHITELIST Internal Hub)"/>
 <arg name="rapp_lists" default="$(optenv RAPP_LIST
     rocon_apps/rocon.rapps;turtlebot_bringup/turtlebot.rapps)"/>
 <arg name="robot_icon" default="turtlebot_bringup/turtlebot2.png"/>
 <arg name="gateway_watch_loop_period" default="10"/> <!-- Polling period for
     multimaster advertising/flipping -->


 <!-- Launch minimal.launch -->
 <remap from="mobile_base/commands/velocity" to="cmd_vel" />
 <include file="$(find turtlebot_bringup)/launch/minimal.launch">
   <arg name="3d_sensor" value="$(arg 3d_sensor)"/>
   <arg name="simulation" value="$(arg simulation)"/>
 </include>

 <!-- Create environment variable for rplidar with "rplidar" default value -->
 <arg name="rplidar_sensor" default="$(optenv TURTLEBOT_LASER_SENSOR rplidar)"/>

 <!-- Set environment variable for rplidar -->
 <include file="$(find turtlebot_le2i)/launch/includes/robot.launch.xml">
   <arg name="base" value="$(arg base)" />
   <arg name="stacks" value="$(arg stacks)" />
   <arg name="rplidar_sensor" value="$(arg rplidar_sensor)"/>
 </include>

 <!-- Launch rplidar -->
 <include file="$(find rplidar_ros)/launch/rplidar.launch"/>

 <!-- paired client for app manager -->
 <include file="$(find rocon_app_manager)/launch/paired_private.launch">
```

```xml
    <arg name="robot_name" value="$(arg robot_name)"/>
    <arg name="robot_type" value="$(arg robot_type)"/>
    <arg name="robot_icon" value="$(arg robot_icon)"/>
    <arg name="rapp_lists" value="$(arg rapp_lists)"/>
    <arg name="gateway_watch_loop_period" value="$(arg gateway_watch_loop_period)"/>
    <arg name="disable_uuids" value="true"/>
    <arg name="hub_whitelist" value="$(arg hub_whitelist)"/>
  </include>
</launch>
```

7 References

1. R. Patrick Goebel, ROS By Example, Version 1.03 for ROS Groovy: April 2013.

2. http://wiki.ros.org/ApplicationsPlatform/Clients/Android/Tutorials/Getting Started

3. Gmapping.http://wiki.ros.org/gmapping!

4. E. M. Eppstain, move base Package Summary, wiki.ros.org, last edited 2012-10-18. http://wiki.ros.org/move base?distro=hydro

5. Willow Garage. Turtlebot. http://www.willowgarage.com/turtlebot, 2008.

6. RPLIDAR, Low Cost 360 degree 2D Laser Scanner (LIDAR) System, Introducion and Datasheet

7. Turtlebot-Android Pairing http://wiki.ros.org/turtlebot android/Tutorials/hydro/Turtlebot-Android%20Pairing

8. Previous ROS Turtlebot report by Deepak Gurung, Jiang Cansen, Jilliam María Farhat, Juan Manuel