Robotics Project: ROS-Turtlebot Motion Control & Navigation

Yanik Porto Mashruf Zaman AK Assad

May 11, 2015

Contents

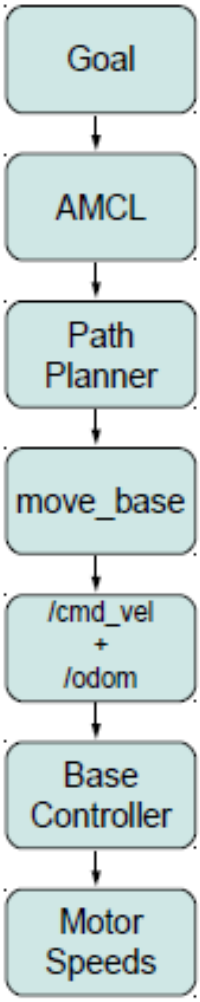
1 Part 1 - Motion Control

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.



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 node and our Workstation as a ROS master. 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_WORKSTATION:11311 >> ~/.bashrc
> echo export ROS_HOSTNAME=IP_OF_TURTLEBOT >> ~/.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 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]

1.6 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
```

But for our case it did find the dock perfectly but with a slight error.

A video of the Auto Docking can be seen in this link:

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

1.7 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 -

1.7.1 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.

1.7.2 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

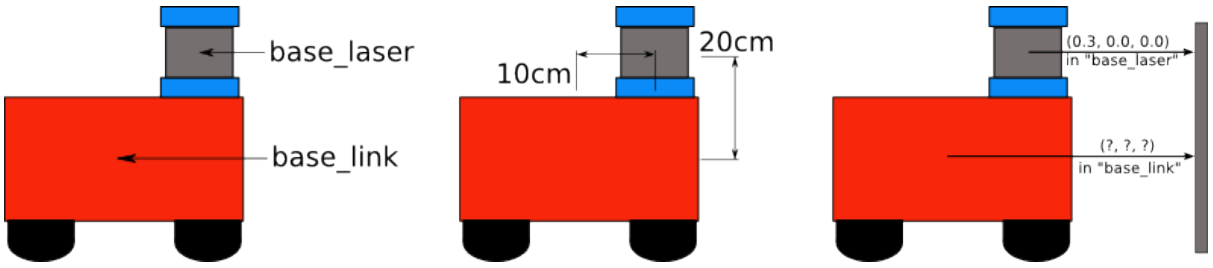
```
$rosrun rbx1_nav timed_out_and_back.py
```

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

1.8 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.



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** 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
```

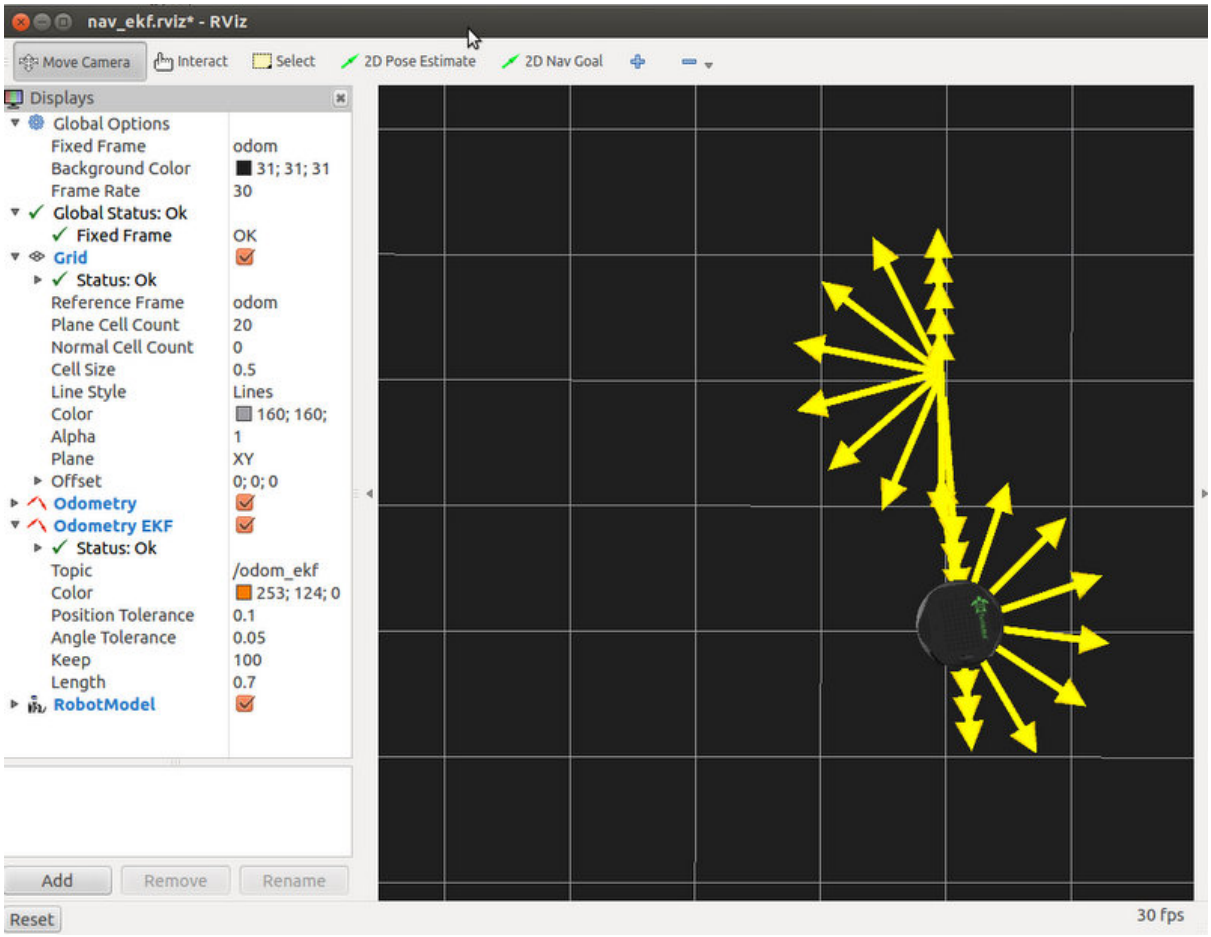
On Workstation: run rviz to see the combined odometry data This is displaying the combined odometry data on the **/odom_ekf** topic rather than just the wheel encoder data published on the **/odom** topic.

```
$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.



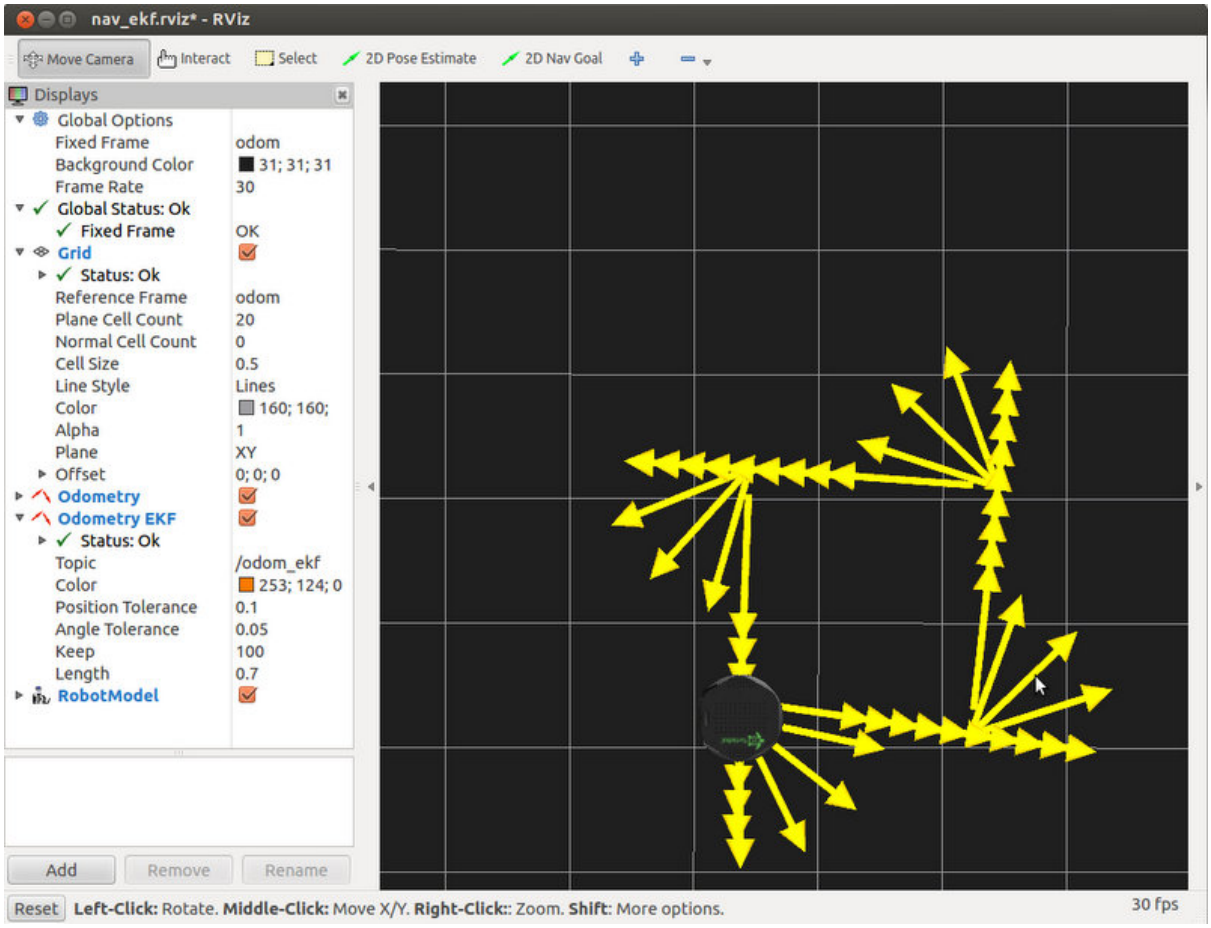
1.8.1 Dead Reckoning

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.

1.9 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.

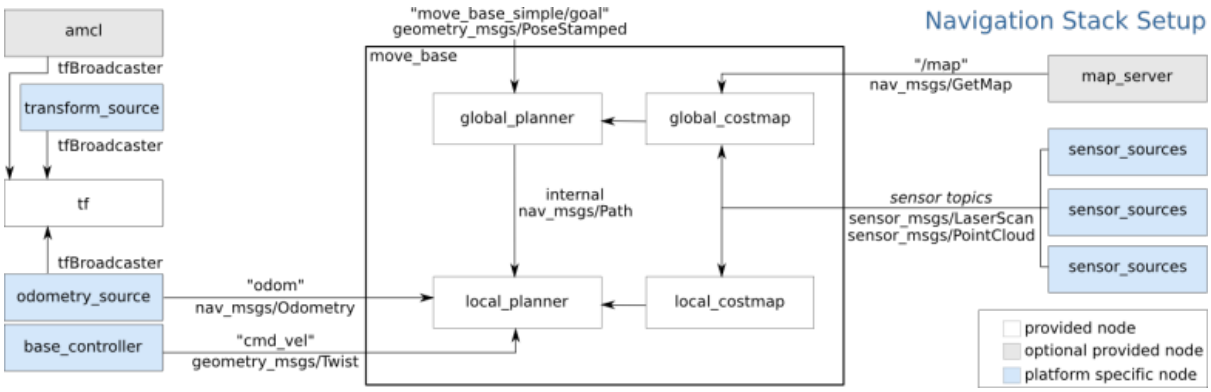
However, this time we will attempt to move the robot in a square by setting four waypoints, one at each corner. At the end of the run, we can see that the errors accumulated in odometry are more visible:

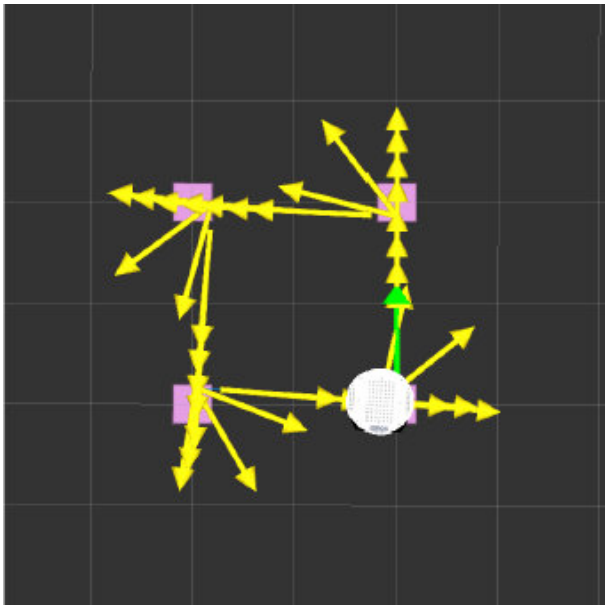


1.10 Goal 4 = Navigation with Path Planning move_base

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.

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.

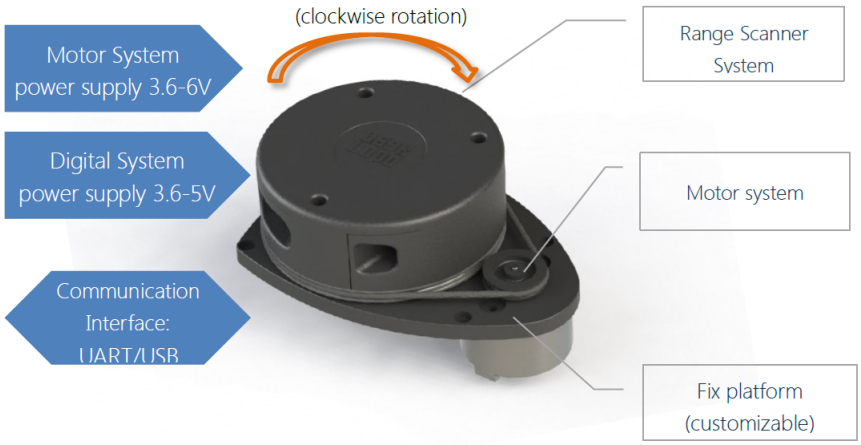




A video of this movement can be seen here:
`https://www.youtube.com/watch?v=5e61iTaRLl0`

2 Part 2 - Planar Laser RangeFinder

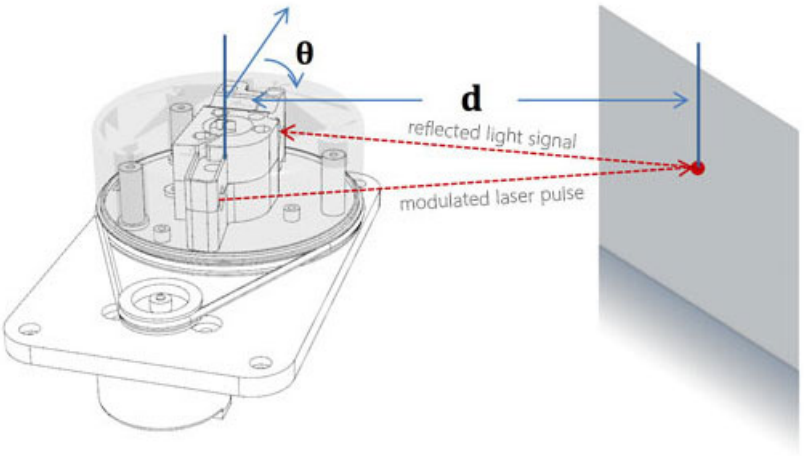
2.1 RP-LIDAR



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.



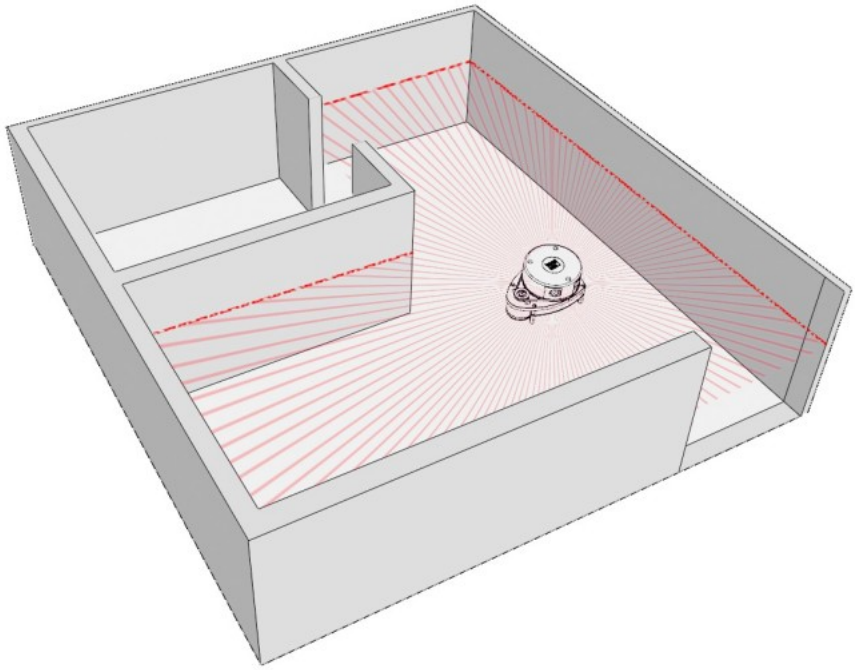
Graphical representation of laser triangulation

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

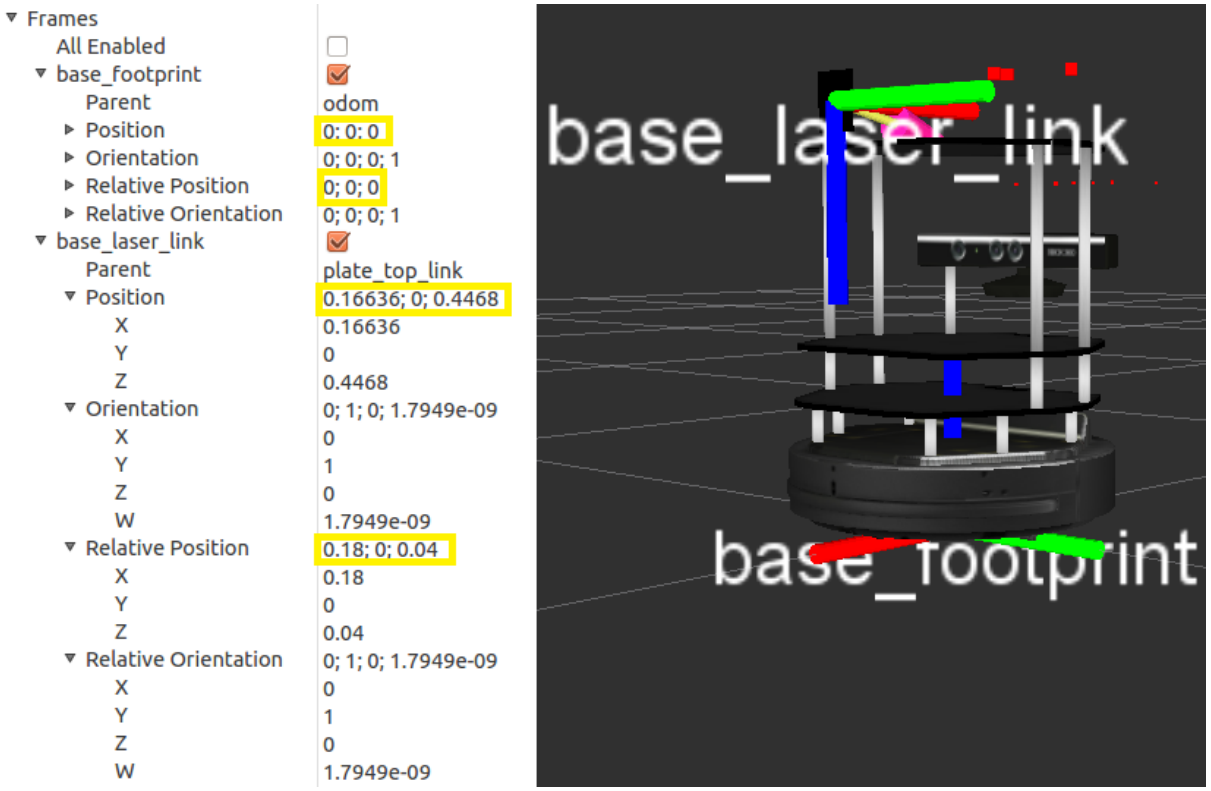
Then it is necessary to create a new package where the position of the Lidar on the turtlebot, compared with the "[parent]" is given (which in our case is the **plate_top_link**). This will be used by the **tf** node, to compute the laser position compared to the base.
In our case, this package is named **"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**.

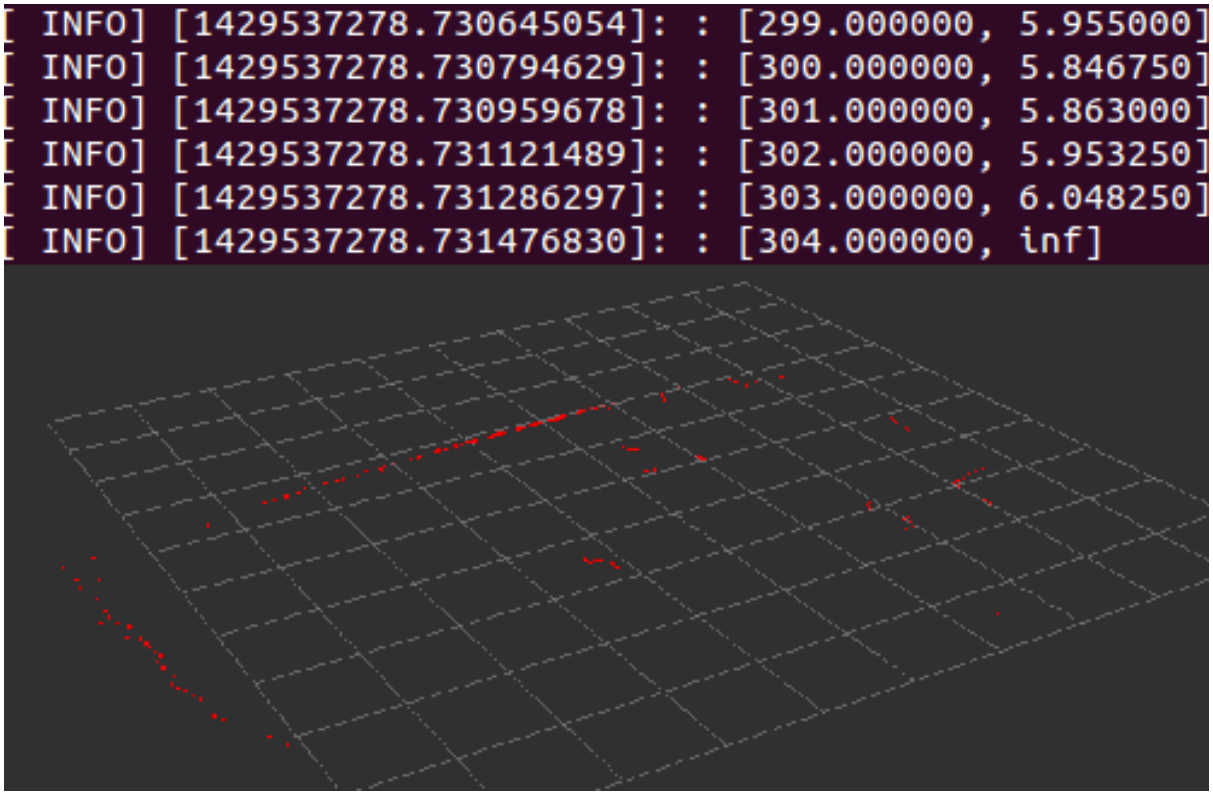
2.2.4 Displaying the data

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

4 Part 4 - Controlling with Android

4.1 Pairing through a public master

4.1.1 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.

4.1.2 Concert

What will be used for controlling with Android is the following statement :

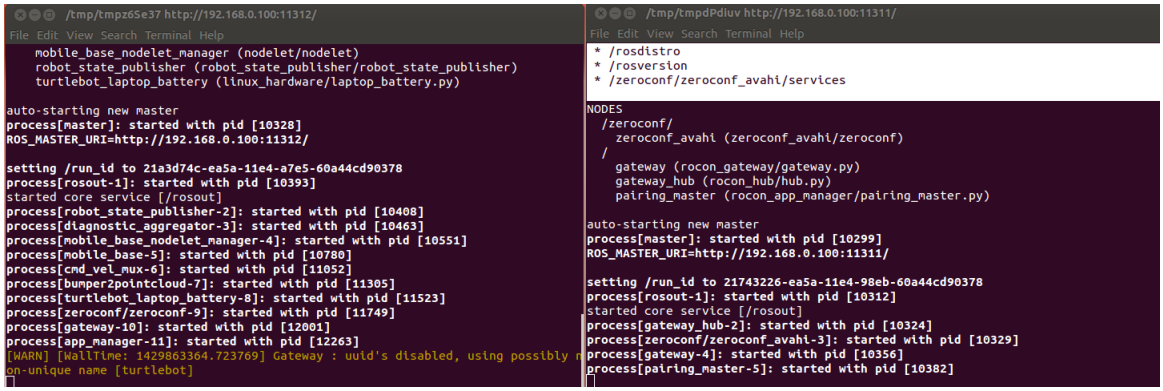
```
$ rocon_launch turtlebot_bringup bringup.concert
```

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.