Autonomous Intelligence System

(Winter Semester: 2020 - 2021)

Topic: Navigation Robotic, Indoor Positioning System

Written by:

Anik Biswas (1324853) anik.biswas@stud.fra-uas.de

Manash Chakraborty (1325085) manash.chakraborty@stud.fra-uas.de

Under the Esteemed Guidance of

Prof. **Dr. Peter Nauth**

Frankfurt University of Applied Sciences

Table of Contents

Abstract

One of the most important targets for mobile robotics is autonomy. Giving a robot the ability to find its own correct location and orientation is one of the most important steps toward achieving this aim. To address this problem, various approaches have been proposed. The functioning of a robot's indoor navigation system using a high-accuracy ultrasonic indoor positioning system is presented in this paper. The Indoor positioning system is comprised of many wireless ultrasonic beacons that are placed in the indoor environment. One beacon is fixed to the robot and mobile, while the others are fixed and have known location coordinates. For demonstration we will be using Turtlebot and Marvelmind Indoor GPS systems.

Introduction:

The positioning is not a new issue. Sextants, clocks, almanacs, charts, and other instruments have been used by mankind to assess his location for millennia. The introduction of the Global Positioning System (GPS), which can provide location information to anybody with a receiver almost anywhere on the planet, is possibly one of the most significant revolutions in this area. GPS, on the other hand, often fails to function in indoor environments because satellite signals lose a lot of power as they pass through walls, making it difficult to receive them with conventional, low-cost sensors. Furthermore, the accuracy of such sensors is limited to a few meters (the Euclidean distance between the true and recorded position), which is inadequate for many indoor applications. As a result, indoor location tracking opens up a plethora of market opportunities. The fact that the indoor mapping market is increasingly growing in size and is expected to be worth about $10 billion by 2020 demonstrates this [1].

Effective robot navigation requires mobile robot localization in an indoor setting. For robot localization, a variety of techniques may be used. These methods should be able to solve both the problem of a missing robot and the problem of a kidnapped robot. In an outdoor environment, global positioning methods can easily solve the problem of a kidnapped robot, but for precise localization, other methods, often based on optical principles and machine vision, such as optical flow or the SLAM algorithm, are needed [2].

Indoor positioning systems (IPSs) have gotten a lot of attention from researchers all over the world in the last few decades for the reasons mentioned above. There are already a number of local positioning systems based on various technologies, each with its own set of benefits and drawbacks. The scanning sweep method is used in systems based on infrared beams, and they are very cheap for consumers, but they only have a precision of 57 cm. IPS employing a visible light contact network has a precision of about 10 cm, but it necessitates costly infrastructure. Ultra-wide-band (UWB) is a great choice for IPS because it has good anti-interference capabilities, but it requires more infrastructure and is costly for users. The highest precision of all technology proposals is IPS based on computer vision, but it is too costly and has limited applications. Radio frequency identification (RFID), WiFi, and ZigBee are the other local positioning solutions, but they have poor positioning accuracy. Ultrasounds have also been used to investigate this problem. The Marvelmind Indoor GPS product uses both ultrasound and radio contact across the ISM radio bands to achieve a precise location of an object within two centimetres precision [3].

For our project we will be integrating Marvelmind with robotic platform Turtlebot. TurtleBot is a ROS standard platform robot. The Robot Operating System (ROS) is a flexible framework for writing robot software. It is a collection of tools, libraries, and conventions that aim to simplify the task of creating complex and robust robot behaviour. Marvelmind supplies ROS package marvelmind_nav, which is able to communicate with mobile beacon or modem and provide received location and other data to the Turtlebot so it could calculate its own location based on available data and move to the desired location.

Hardware Information

On our project we have used mainly two set of hardware platform.

1. Turtelbot3 Burger
2. Marvelmind Indoor GPS

Turtelbot3 Burger

The TurtleBot is a low-cost, open-source personal robot kit. Melonee Wise and Tully Foote built TurtleBot in November 2010 at Willow Garage. The TurtleBot can create a map using SLAM (simultaneous localization and mapping) algorithms and drive around a space. It can also be operated remotely using a laptop, joypad, or Android phone. We used Turtlebot3 Burger for our project. It includes mainly four major components [4].

1. Raspberry Pi 3 Model B Single Board Computer (SBC)
2. Laser Distance Sensor
3. OpenCR control board
4. Dynamixel X series actuators

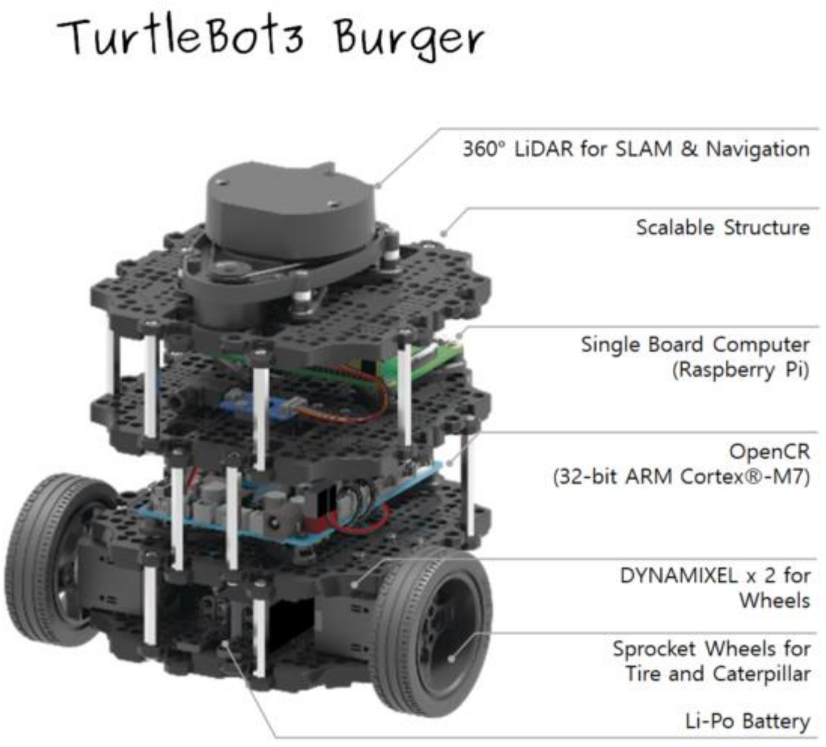

TurtleBot3 Burger
360° LiDAR for SLAM & Navigation
Scalable Structure
Single Board Computer (Raspberry Pi)
OpenCR (32-bit ARM Cortex®-M7)
DYNAMIXEL x 2 for Wheels
Sprocket Wheels for Tire and Caterpillar
Li-Po Battery

Figure 1: TurtleBot3 Burger

Marvelmind Indoor GPS

For the navigation and tracking system we are using Marvelmind Indoor Navigation System. It is an off-the-shelf indoor navigation system, designed to provide precise location data to autonomous robots. The main benefit of this system is, it has a GUI based application, which allows automatic distance measurement of the beacons, automatic beacons map forming and some other features for commissioning and development [5].

On our project we are using Starter Set HW v4.9 NIA. It contains four stationary beacons, one mobile beacon and one modem with recommended distance between beacons 30m and coverage area of 1000m2. The indoor positioning system by Marvelmind robotics uses ultrasound ranging to determine the position of the mobile beacon module (referred to as hedgehogs). Ultrasound ranging is also used by the beacons to determine their relative position. Therefore, the Marvelmind system is self-calibrating. The sensor modules have built-in rechargeable batteries. Mobile beacon's location is calculated based on a propagation delay of an ultrasonic pulses (Time-Of-Flight or TOF) between stationary and mobile beacons using trilateration algorithm [5]. The benefit of ultrasonic wave is that it is slower than radio wave, so the time synchronization can be made by radio transceiver.

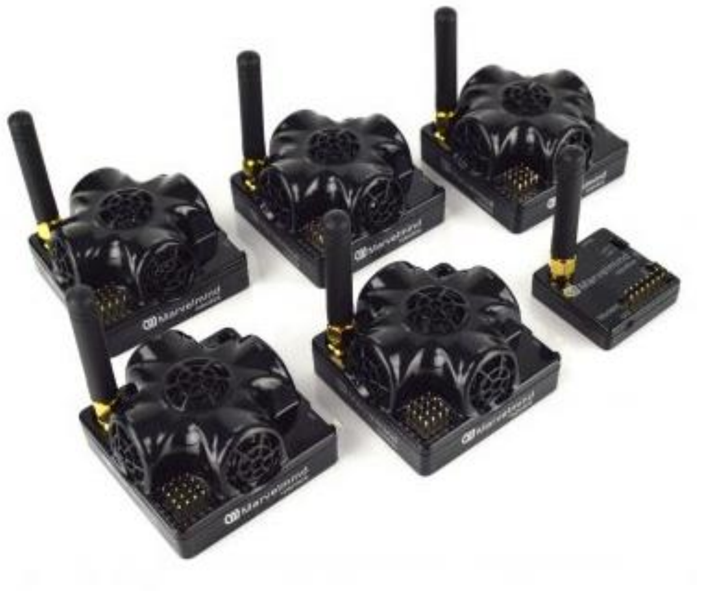


Figure 2: Marvelmind Starter Set HW v4.9-NIA

The following is an explanation of the trilateration principle. If we calculate the TOF, t_k, to each stationary beacon k from the mobile beacon at an unknown location (x, y, z), then multiply them by the speed of the propagating wave, c, we get a set of ranges from the mobile beacon to each individual stationary beacon, r_k. Considering that mobile beacon and stationary beacons are synchronized, i.e., their clocks have exactly the same time without major delays, we get a set of distances from the mobile beacon to each individual stationary beacon. From this writing down the equation of a sphere of radius r_k cantered at each stationary beacon at position $(x_k,\ y_k,\ z_k)$ (eq. 1), we obtain a system of N equations $(N \geq 3)$, whose solution tell us the receiver or user position (x, y, z) [6].

$$\sqrt{(x_k - x)^2 + (y_k - y)^2 + (z_k - z)^2} = r_k$$

Software Platform

On our project we have used mainly 3 software platform.

1. Raspberry Pi OS
2. Ubuntu 16.04 (Xenial Xerus)
3. ROS (Robot Operating System)

Raspberry Pi OS

Raspberry Pi OS (previously known as Raspbian) is a Debian-based operating system for Raspberry Pi. It has been the standard operating system for the Raspberry Pi family of small single-board computers since 2015.

Ubuntu 16.04 (Xenial Xerus)

Ubuntu is a Linux distribution based on Debian and composed mostly of free and open-source software. On our project we are using Ubuntu 16.04 (Xenial Xerus). It was released in 2016 and it is used on our project Marvelmind system is supported on ROS Kinetic and it is build and best supported on Ubuntu 16.04.

ROS (Robot Operating System)

ROS is an open-source, meta-operating system for a robot. It provides the services which would be expect from an operating system, including hardware abstraction, low-level device control, implementation of commonly used functionality, message-passing between processes, and package management. It also provides tools and libraries for obtaining, building, writing, and running code across multiple computers. We have discussed about some of the basic components of the ROS below.

ROS master

For node-to-node connections and message exchange, the master serves as a name server. The master is run with the command roscore. Without the master, it is impossible to connect nodes and communicate messages such as topics and services. At startup every node needs to get registered with the master. The master communicates with slaves using XMLRPC (XML-Remote Procedure Call), which is an HTTP-based protocol that does not maintain connectivity. The slave nodes can access only when they need to register their own information or request information of other nodes. The connection status of each other is not checked regularly. Due to this feature, ROS can be used in very large and complex environments [7].

ROS Node

A node refers to the smallest unit of processor running in ROS. It is recommended to create one single node for one purpose. Each node acts separately from each other, so they are individually compiled, executed and managed. Each node will register with the master at startup, providing information such as the node's name, message type, URI address, and port number. Based on the registered information, the registered node can act as a publisher, subscriber, service server, or service client, and nodes can exchange messages utilizing topics and services. Packages are used to organize nodes. [7].

ROS Message

ROS is developed in unit of nodes, which is the minimum unit of executable program that has broken down for the maximum reusability. The node exchanges data with other nodes through messages forming a large program as a whole. The key concept here is the message communication methods among nodes. There are three different methods of exchanging messages: a topic which provides a unidirectional message transmission/ reception, a service which provides a bidirectional message request/response and an action which provides a bidirectional message goal/result/feedback [7].

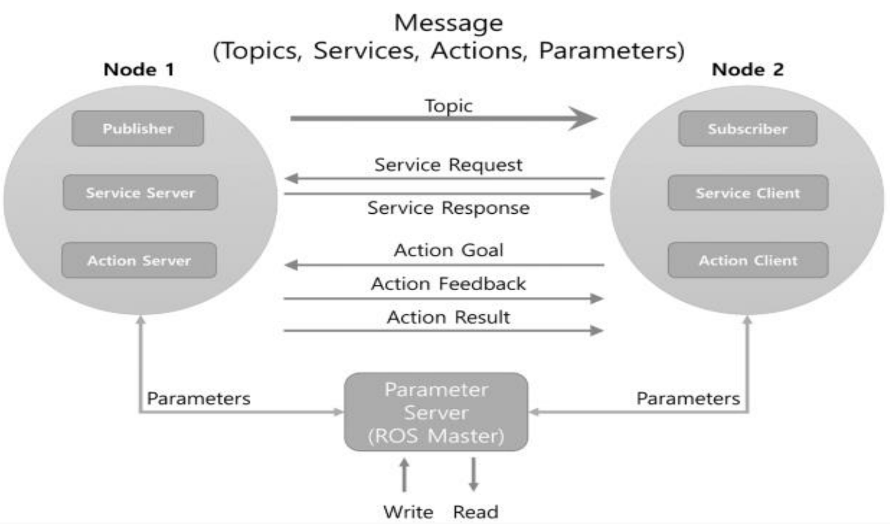


Figure 3: Message Communication between Nodes

Type	Features	Description	
Topic	Asynchronous	Unidirectional	Used when exchanging data continuously
Service	Synchronous	Bi-directional	Used when request processing requests and responds current state
Action	Asynchronous	Bi-directional	Used when it is difficult to use the service due to long response times after the request or when an intermediate feedback value is needed.

Table 1: Comparison of the Topic, Server, and Action

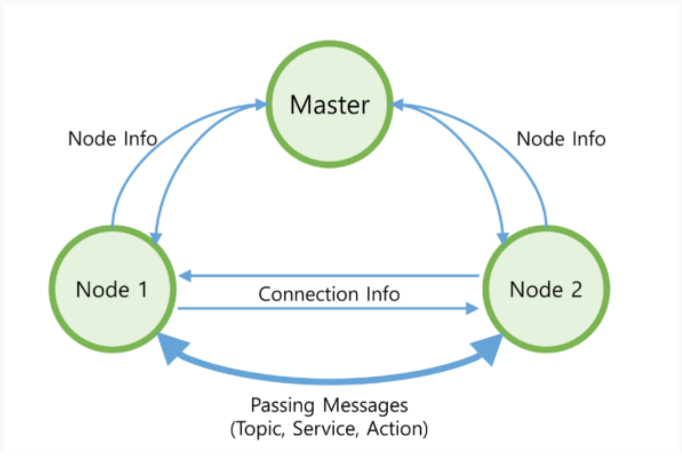


Figure 4: Message Communication

ROS Topics

Nodes in ROS interact using topics. A topic may be published or subscribed to by a node. After registering its topic with the master, the publisher node begins publishing messages on that topic. Subscriber nodes who want to receive the subject send a request to the publisher node, referencing the topic name registered with the master. The subscriber node connects directly to the publisher node using this information to exchange messages as a topic. A publisher and subscriber are declared in the node and can be declared multiple times in one node. Topics are suitable for sensor data that involves publishing messages on a regular basis since they are unidirectional and remain linked to continuously send or receive messages. Furthermore, a publisher can send messages to multiple subscribers [7].

ROS Service

A bidirectional synchronous communication between the service client requesting a service and the service server responding to the request is known as service communication. A service consists of a service server that only responds when a request is made and a service client that can both send and receive requests. The service, unlike the topic, is a one-time message communication. As a result, once the service's request and response have been completed, the link between the two nodes will be severed [7].

ROS Actions

Another message exchange method for asynchronous bidirectional communication is the action. When responding to a request takes longer than expected and intermediate responses are needed before the answer is returned, action is used [7].

In our project we have used ROS Kinetic Kame. ROS Kinetic Kame is primarily targeted at the Ubuntu 16.04 (Xenial) release [8].

Figure 5: ROS Kinetic Kame

Localization

In a dynamic, unstructured world, achieving high-precision and high-reliability localization and sometimes building a map remains a difficult challenge. The mobile robot's operating environment is usually set for a period of time in an industrial realistic application scenario. In this static setting, probabilistic localization methods have been shown to be reliable. The extended Kalman filter provides the basis for the probabilistic localization approach. histogram filter, and particle filter, namely Monte Carlo localization (MCL) [9].

In our implementation of localization, we have used Adaptive Monte Carlo localization (AMCL) and SLAM (Simultaneous Localization and Mapping).

AMCL

MCL, also known as particle filter localization, is a robot localization algorithm that uses a particle filter. The algorithm calculates the location and orientation of a robot as it moves and senses the environment using a map of the environment. The algorithm represents the distribution of likely states using a particle filter, with each particle representing a possible state [10].

The purpose of the algorithm is for the robot to determine its position within the environment given a map of the environment.

At every time t the algorithm takes as input the previous belief (The *belief*, which is the robot's estimate of its current state) $X_{t-1} = \{X_{t-1,}^{[1]} X_{t-1,}^{[2]} X_{t-1,}^{[3]} \dots \dots \dots X_{t-1,}^{[M]}\}$, an actuation command u_t, and data received from sensors z_t; and the algorithm outputs the new belief X_t [10].

Algorithm: Monte Carlo Localization

```
1:   **procedure** MCL ($X_{t-1}, u_t, z_t, m$)
2:       $\bar{\boldsymbol{X}}_t = \boldsymbol{X}_t = \emptyset$
3:       for $m = 1$ to $M$ do
4:           $x_t^{[m]}$ = sampleMotionModel ($u_t, x_{t-1}^{[m]}$)
5:           $m_t^{[m]}$ = measurementModel ($z_t, x_t^{[m]}, m$)
6:           $\bar{X}_t = \bar{X}_t + < x_t^{[m]}, w_t^{[m]} >$
7:       **for** $m = 1$ to $M$ **do**
8:           draw $x_t^{[m]}$ from $\bar{\boldsymbol{X}}_t$ with probability $\propto w_t^{[m]}$
9:           $\boldsymbol{X}_t = \boldsymbol{X}_t + x_t^{[m]}$
10:      **return** $\boldsymbol{X}_t$
```

The AMCL algorithm is an adaptation of the MCL algorithm that is much better suited to solving the kidnapped robot problem than the MCL. An error estimate determines the number of particles used in AMCL. Because of this development, the algorithm now uses a large number of particles when the estimation output is low and a small number of particles when the particles converge nicely [9]. KLD (Kullback-Leibler divergence) sampling is the particle filter used in Adaptive Monte Carlo Localization. The term adaptive is used in Adaptive Monte Carlo Localization because of KLD sampling. Based on the uncertainty, the KLD sampling method selects the number of samples for the sampling procedure. The greater are the degree of uncertainty, the greater are the number of samples chosen, and vice versa [11].

SLAM

The SLAM (Simultaneous Localization and Mapping) problem is one of the most challenging challenges to overcome when developing autonomous capabilities for mobile vehicles, according to the robotics literature. The aim of SLAM is for an autonomous vehicle to start from an unknown location in an unknown environment and incrementally create a map of its environment using the uncertain information derived from its sensors, while also using that map to localize itself in relation to a reference coordinate frame and navigate in real-time.

In our project we are using the ROS platform. The gmapping package will be used to create SLAM in ROS. A ROS wrapper for OpenSlam's Gmapping is included in the GMapping package. The gmapping package provides laser-based SLAM, as a ROS node called slam_gmapping. We can generate a 2D occupancy grid map (similar to a building floorplan) from laser and pose data collected by a mobile robot using slam_gmapping. The slam_gmapping node receives sensor msgs/LaserScan messages and creates a map (nav_msgs/OccupancyGrid) based on them. The map can be retrieved via a ROS topic or service [12].

Kidnapped Robot problem

As mobile robot localization approaches continue to improve, it has become imperative to address more complicated challenges in this domain such as the Kidnapped Robot Problem. The kidnapped robot problem in robotics refers to a circumstance in which an operating autonomous robot is taken to an unknown location and the challenge is to estimate its position by application of one or more navigation technique. The challenges involved with the kidnapped robot causes considerable problems with the robot's localization mechanism.

Robot localization using the MCL algorithm, and a Laser Range Finder has great accuracy but fails when the problem of kidnapped robot arises, but robot localization using the MCL method as well as an indoor GPS system is resistant to the problem of kidnapped robot. One of the advantages of adopting an indoor GPS system is that the location of the robot can now be sampled directly from the sensor's models and current sensor measurements, allowing for quick global localization. As a result, even when the mobile robot is kidnapped, the robot can always be located as long as it is within the range of the ultrasound signal coverage [13].

Implementation [14] :

Localization of Turtlebot in ROS platform:

Hardware Assembly: Turtelbot3 Burger assembly done as per the instructions given on the manual provided by Robotis. Final constructed unit is in figure 6. The following components were used for the construction for the same: Waffles, OpenCr Board, Rapsberry Pi 3B+, Power connectors, Dymnamxel, Lithium-ion battery

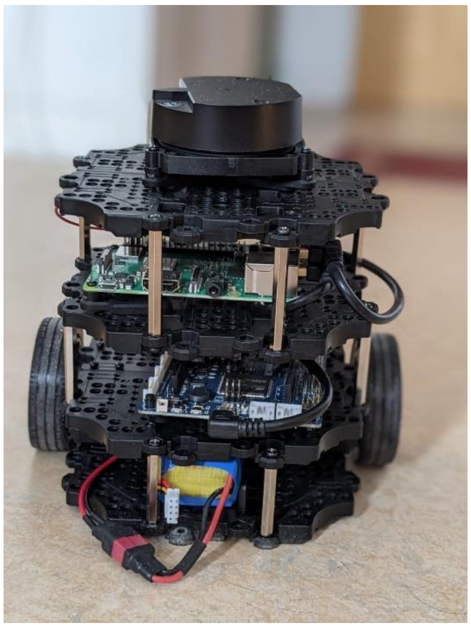


Figure 6: Assembled Turtelbo3 burger

Rapsberry Pi 3 B+: Rapsberry Pi OS is installed in the Rapsberry Pi device and network configuration is done so that it can connect with our remote PC/ Ubuntu base machine.

ROS Installation: In the Remote PC we installed ROS Kinetic on Ubuntu 16.04 platform. Subsequently the following packages are installed:

*ros-kinetic-joy ros-kinetic-teleop-twist-joy *
*ros-kinetic-teleop-twist-keyboard ros-kinetic-laser-proc *
*ros-kinetic-rgbd-launch ros-kinetic-depthimage-to-laserscan *
*ros-kinetic-rosserial-arduino ros-kinetic-rosserial-python *
*ros-kinetic-rosserial-server ros-kinetic-rosserial-client *
*ros-kinetic-rosserial-msgs ros-kinetic-amcl ros-kinetic-map-server *
*ros-kinetic-move-base ros-kinetic-urdf ros-kinetic-xacro *
ros-kinetic-compressed-image-transport ros-kinetic-rqt *
ros-kinetic-gmapping ros-kinetic-navigation ros-kinetic-interactive-markers
ros-kinetic-dynamixel-sdk
ros-kinetic-turtlebot3-msgs
ros-kinetic-turtlebot3

ROS Master-Node: ROS Master is established in the Remote Ubuntu PC by editing the .bashrc file. ROS Nodes can communicate through established computational methods.

```
export ROS_MASTER_URI =http://192.168.0.216:11311
export ROS_HOSTNAME = 192.168.0.216
```

Roscore started on the remote PC which is acting as ROS master.

OpenCR Setup: The OpenCR 1.0 firmware and packages are installed through the Rapsberry Pi connection. The OpenCR board switches can be used manually to check performance of Dynamixel (Robot Motors) and Laser Scanner devices.

Turtlebot Bringup and Movement Control: Once all setups are complete, we can launch the Turtlebot with the following command.

```
roslaunch turtlebot3_bringup turtlebot3_robot.launch
```

We can use the teleop key features to control movement and direction of the turtlebot and calibrate speed.

```
export TURTLEBOT3_MODEL=${TB3_MODEL}
roslaunch turtlebot3_teleop turtlebot3_teleop_key.launch
```

ROS SLAM: SLAM (Simultaneous Localization and Mapping) is the process of drawing a map by estimating the current location in an arbitrary map space.The ROS SLAM package turtlebot3_slam had already been installed. The following command launches the same in the service:

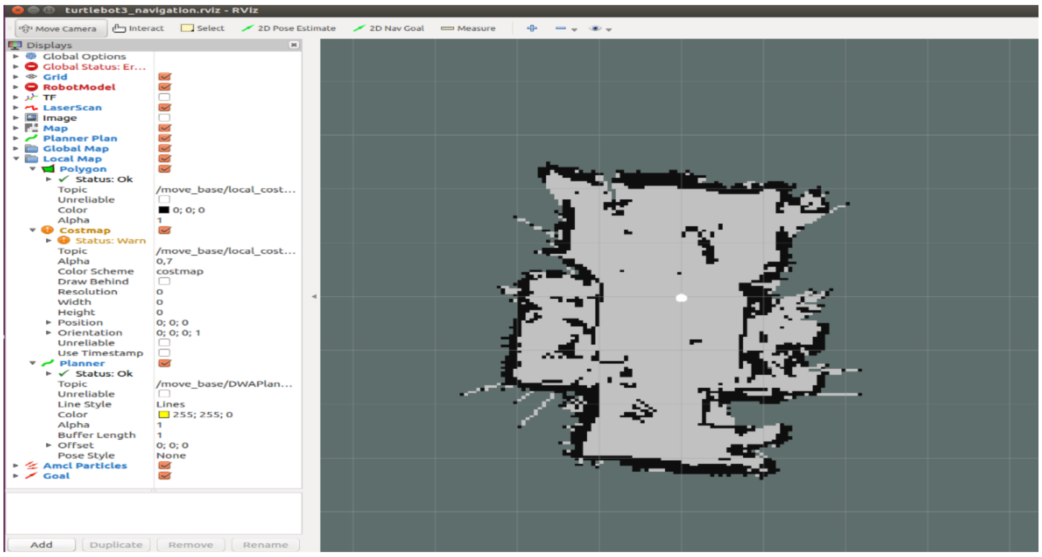
```
roslaunch turtlebot3_slam turtlebot3_slam.launch
```

Map Service and Save map: MAP service is launched with the following command ,

```
map_server map_saver -f ~/map
```

While the Map service is ON, the robot is moved through the environment with teleop key. The laser at the top of Turtelbot3 scans the environment the Map is stored in a yaml file in the home directory.

```
roslaunch  turtlebot3_navigation turtlebot3_navigation.launch map_file:=$HOME/map.yaml
```



Global Options	
Global Status: Er...	
Grid	✓
RobotModel	✓
TF	☐
LaserScan	✓
Image	☐
Map	✓
Planner Plan	✓
Global Map	✓
Local Map	✓
Polygon	✓
Status: Ok	
Topic	/move_base/local_cost...
Unreliable	☐
Color	■ 0; 0; 0
Alpha	1
Costmap	✓
Status: Warn	
Topic	/move_base/local_cost...
Alpha	0,7
Color Scheme	costmap
Draw Behind	☐
Resolution	0
Width	0
Height	0
Position	0; 0; 0
Orientation	0; 0; 0; 1
Unreliable	☐
Use Timestamp	☐
Planner	✓
Status: Ok	
Topic	/move_base/DWAPlan...
Unreliable	☐
Line Style	Lines
Color	🟨 255; 255; 0
Alpha	1
Buffer Length	1
Offset	0; 0; 0
Pose Style	None
Amcl Particles	✓
Goal	✓

Figure 7: Room map build by Turtelbot3

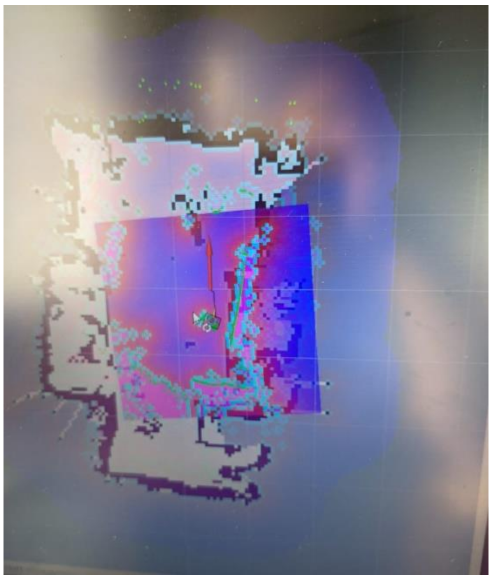


Figure 8: Turtelbot3 navigating based on stored map

Navigation through Stored Map : Navigation is the process of moving to a goal with specific co-ordinates. In this process, the map stored earlier in the SLAM process is used for navigation purpose.

```
roslaunch turtlebot3_navigation turtlebot3_navigation.launch
map_file:=$HOME/map.yaml
```

Once the above command is launched RViz menu is opened where the realtime map-based position of the robot is displayed. It is important to align initial position of the turtlebot for this this purpose. As an approach to solve the localization problem, this map-based navigation of the turtlebot provides solution for both obstacle avoidance and moving to specific goal point. Setting up instructions to move to a specific point can be implemented both through the RViz emulator as well as a python script which is included in the Appendix section. The script uses ROS Action Server and Action client architecture and includes several ROS messages like `MoveBaseAction` and

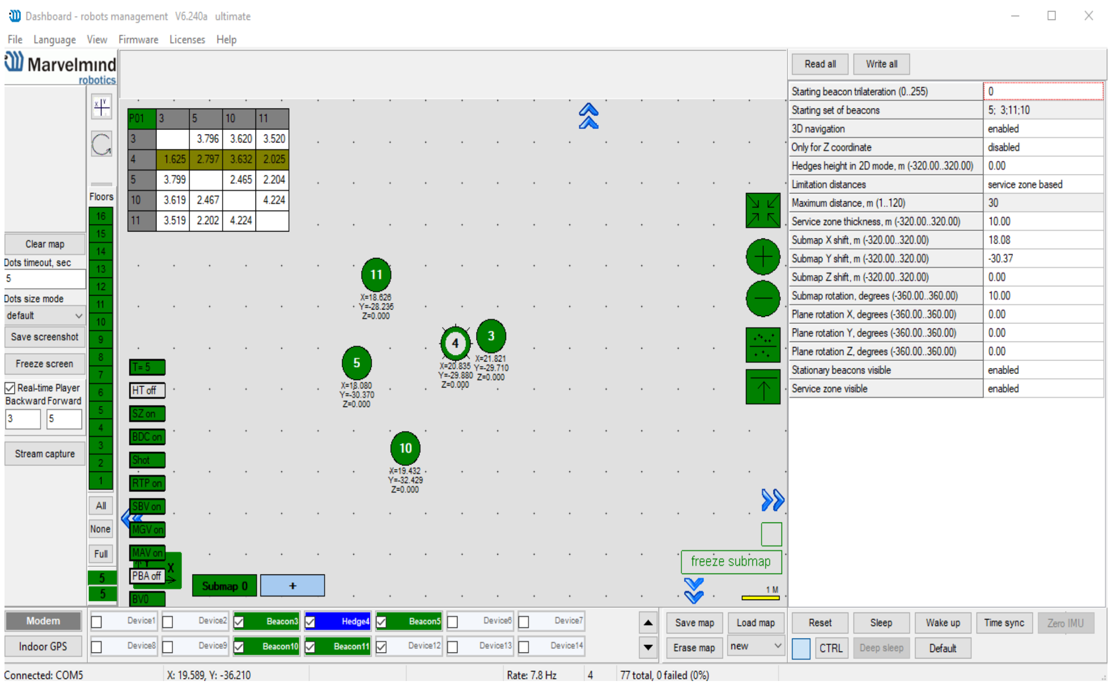
`MoveBaseGoal` to perform the navigation task as well as the `geometry_msgs` to orient position and direction.

Marvelmind Indoor GPS Sensor:

As discussed earlier, the Marvelmind starter kit consists of one Modem, four Stationary Beacons and one Mobile Beacon. The following steps are used to set up the Indoor GPS system:

Step 1: Marvelmind Software pack is downloaded and from that the Dashboard software is installed. In our case the Dashboard pack is used for Linux variant which is designed for x86 architecture (Modem HW 4.9 to be connected to Ubuntu Remote PC).

Step 2: All the beacons are charged and connected to the PC containing Dashboard software. Specific Firmware are loaded onto the devices. Similarly, firmware is loaded for the modem.



P01	3	5	10	11
3		3.796	3.620	3.520
4	1.625	2.797	3.632	2.025
5	3.799		2.465	2.204
10	3.619	2.467		4.224
11	3.519	2.202	4.224	

Starting beacon trilateration (0..255)	0
Starting set of beacons	5; 3;11;10
3D navigation	enabled
Only for Z coordinate	disabled
Hedges height in 2D mode, m (-320.00..320.00)	0.00
Limitation distances	service zone based
Maximum distance, m (1..120)	30
Service zone thickness, m (-320.00..320.00)	10.00
Submap X shift, m (-320.00..320.00)	18.08
Submap Y shift, m (-320.00..320.00)	-30.37
Submap Z shift, m (-320.00..320.00)	0.00
Submap rotation, degrees (-360.00..360.00)	10.00
Plane rotation X, degrees (-360.00..360.00)	0.00
Plane rotation Y, degrees (-360.00..360.00)	0.00
Plane rotation Z, degrees (-360.00..360.00)	0.00
Stationary beacons visible	enabled
Service zone visible	enabled

Connected: COM5 X: 19.589, Y: -36.210 Rate: 7.8 Hz 4 77 total, 0 failed (0%)

Figure 9: Marvelmind dashboard showing Beacon positions

Step 3: The modem is connected to the Dashboard PC. One after another, all the Beacons are placed. In our approach we would be using Non-Inverse Architecure. The beacons are added in the Dashboard map and a distance calculation matrix is displayed in the Dashboard. It is important to place all the beacons in a line of sight with the Modem to accurately create the MAP. If there are more than four beacons in the Map, then submap need to be created and the additional beacons to be integrated in that.

Marvelmind Device and ROS Integration:

Since one of the core project idea is to accurately predict turtlebot position with the help of Marvelmind sensor, the generated navigation data has to be received and processed by the ROS system. The mobile beacon is to be connected to the Rapsberry Pi/ Turtlebot and GPS data would be processed by the ROS node. The package marvelmind_nav is used for this purpose. In the Rapsberry Pi system the package is installed, and the catkin workspace is rebuilt.

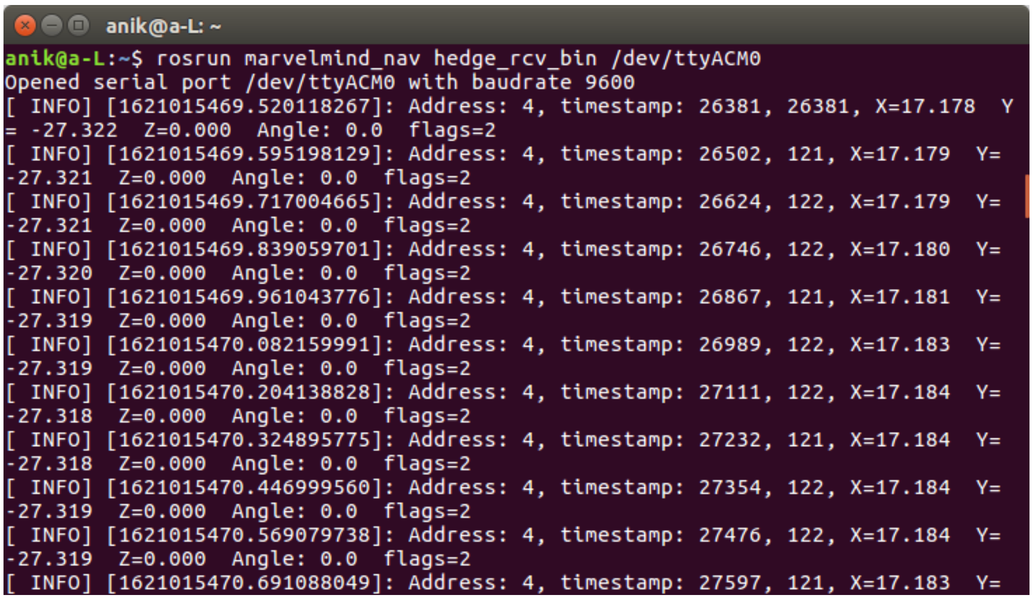
The mobile beacon is connected to the Rapsberry Pi through a USB and cable and UART is enabled in the Serial to process the sensor communication.

Roscore is run in the Rapsberry pi.

In our system the serial port used for communication is /dev/ttyACM0. Hence we launch the marvelmind package hedge_rcv_bin' with the following command

```
rosrun marvelmind_nav hedge_rcv_bin /dev/ttyACM0
```

The following positional data can be observed:



```
anik@a-L:~$ rosrun marvelmind_nav hedge_rcv_bin /dev/ttyACM0
Opened serial port /dev/ttyACM0 with baudrate 9600
[ INFO] [1621015469.520118267]: Address: 4, timestamp: 26381, 26381, X=17.178  Y
= -27.322  Z=0.000  Angle: 0.0  flags=2
[ INFO] [1621015469.595198129]: Address: 4, timestamp: 26502, 121, X=17.179  Y=
-27.321  Z=0.000  Angle: 0.0  flags=2
[ INFO] [1621015469.717004665]: Address: 4, timestamp: 26624, 122, X=17.179  Y=
-27.321  Z=0.000  Angle: 0.0  flags=2
[ INFO] [1621015469.839059701]: Address: 4, timestamp: 26746, 122, X=17.180  Y=
-27.320  Z=0.000  Angle: 0.0  flags=2
[ INFO] [1621015469.961043776]: Address: 4, timestamp: 26867, 121, X=17.181  Y=
-27.319  Z=0.000  Angle: 0.0  flags=2
[ INFO] [1621015470.082159991]: Address: 4, timestamp: 26989, 122, X=17.183  Y=
-27.319  Z=0.000  Angle: 0.0  flags=2
[ INFO] [1621015470.204138828]: Address: 4, timestamp: 27111, 122, X=17.184  Y=
-27.318  Z=0.000  Angle: 0.0  flags=2
[ INFO] [1621015470.324895775]: Address: 4, timestamp: 27232, 121, X=17.184  Y=
-27.318  Z=0.000  Angle: 0.0  flags=2
[ INFO] [1621015470.446999560]: Address: 4, timestamp: 27354, 122, X=17.184  Y=
-27.319  Z=0.000  Angle: 0.0  flags=2
[ INFO] [1621015470.569079738]: Address: 4, timestamp: 27476, 122, X=17.184  Y=
-27.319  Z=0.000  Angle: 0.0  flags=2
[ INFO] [1621015470.691088049]: Address: 4, timestamp: 27597, 121, X=17.183  Y=
```

Fusion of Marvelmind Sensor Data with ROS Localization:

Since the Turtlebot Laser scanner data is not accurate enough to navigate the map data it would be helpful to integrate the GPS data with the ros_localization data to achieve precision and accuracy. Fusing the GPS data for the hedgehog mobile beacon position with the ROS would be a good idea to realize the above objective. One useful approach for this kind of sensor fusion would be Kalman Filter or Extended Kalman Filter tuning. The primary mechanism in this regard would be to calibrate AMCL with the Beacon position data through some technique of transformations.

Appendix I

Python Script for Turtelbot navigation

```
#!/usr/bin/env python
import rospy
import actionlib
from move_base_msgs.msg import MoveBaseAction, MoveBaseGoal
from math import radians, degrees
from actionlib_msgs.msg import *
from geometry_msgs.msg import Point

#Method for moving the robot to destination

def go_to_destination(xGoal,yGoal):

   #Defining Action Client

   ac = actionlib.SimpleActionClient("move", MoveBaseAction)

   #Waiting fro response from action server

   while(not ac.wait_for_server(rospy.Duration.from_sec(5.0))):
           rospy.loginfo("Waiting for the move_base action server to
respond")

   goal = MoveBaseGoal()
   #Frame parameters
   goal.target_pose.header.frame_id = "map"
   goal.target_pose.header.stamp = rospy.Time.now()

   # moving towards the destination*/

   goal.target_pose.pose.position =  Point(xGoal,yGoal,0)
   goal.target_pose.pose.orientation.x = 0.0
   goal.target_pose.pose.orientation.y = 0.0
   goal.target_pose.pose.orientation.z = 0.0
   goal.target_pose.pose.orientation.w = 1.0

   rospy.loginfo("Calibrating destination co-ordinate")
   ac.send_goal(goal)

   ac.wait_for_result(rospy.Duration(60))

   if(ac.get_state() ==  GoalStatus.SUCCEEDED):
           rospy.loginfo("Robot reached destination")
           return True
   else:
           rospy.loginfo("Failed to reach destination")
           return False
if __name__ == '__main__':
   rospy.init_node('map_navigation', anonymous=False)
   x_goal = float(input("Set goal in x co-ordinate:"))
   y_goal = float(input("Set goal in x co-ordinate:"))

   go_to_destination(x_goal,y_goal)
   rospy.spin()
```

References

[1] R. Amsters, E. Demeester, N. Stevens, Q. Lauwers and P. Slaets, “Evaluation of low-cost/high-accuracy indoor positioning systems,” in *Proceedings of the 2019 International Conference on Advances in Sensors, Actuators, Metering and Sensing (ALLSENSORS)*, Athens, 2019.

[2] J. Černohorský, P. Jandura and a. P. Rydlo, “Real time ultra-wideband localisation,” *19th International Carpathian Control Conference (ICCC),* pp. 445-450, 2018.

[3] J. Qi and G.-P. Liu., “A robust high-accuracy ultrasound indoor positioning system based on a wireless sensor network,” *Sensors,* vol. 17, p. 2554, 2017.

[4] “https://emanual.robotis.com/docs/en/platform/turtlebot3/features/,” [Online].

[5] M. Robotics, *Marvelmind indoor navigation system operating manual.,* 2017.

[6] A. Jimenez and F. Seco, “Ultrasonic localization methods for accurate positioning,” *Instituto de Automatica Industrial,* 2005.

[7] Y. Pyo, H. Cho, L. Jung and D. Lim, ROS Robot Programming (English), ROBOTIS, 2017.

[8] “ROS Kinetic Kame,” [Online]. Available: http://wiki.ros.org/kinetic.

[9] G. Peng, W. Zheng, Z. Lu, J. Liao, L. Hu, G. Zhang and D. He, “An improved AMCL algorithm based on laser scanning match in a complex and unstructured environment,” *Complexity,* 2018.

[10] S. Thrun, W. Burgard and D. Fox, Probabilistic robotics, MIT press, 2005.

[11] R. Mishra and A. Javed, “ROS based service robot platform,” *4th International Conference on Control, Automation and Robotics,* pp. 55--59, 2018.

[12] “http://wiki.ros.org/gmapping,” [Online].

[13] Y. Seow, R. Miyagusuku, A. Yamashita and H. Asama, “Detecting and solving the kidnapped robot problem using laser range finder and wifi signal,” *IEEE international conference on real-time computing and robotics (RCAR),* pp. 303--308, 2017.

[14] [Online]. Available: https://emanual.robotis.com/docs/en/platform/turtlebot3/overview/#overview.