ROS Based Control and Simulation Framework for Obstacle Avoidance using a Two Wheeled Robot

J COMPONENT REPORT

by

RASLAN MOHAMED

18BEC0747

MOHIT SV

18BEC0878

SHIMONI MEHROTRA

18BEC0884

JOSHUA ALWIN

18BEC0986

MAHAK CHOPRA

18BEC0992

School of Electronics
Vellore Institute of Technology
Vellore

November 2020

Declaration by Authors

This is to declare that this report has been written by us as part of our coursework. No part of the report is plagiarized from other sources. All information included from other sources have been duly acknowledged. We aver that if any part of the report is found to be plagiarized, we shall take full responsibility for it.

Raslan Mohamed
18BEC0747

Mohit SV
18BEC0878

Shimoni Mehrotra
18BEC0884

Joshua Alwin
18BEC0986

Mahak Chopra
18BEC0992

Place: VIT Vellore

Date: 30th October, 2020

Abstract

The problem we addressed was how our knowledge about robotics related software isn't much. We wanted to understand the software ROS and use it to carry out simulation based experiments. However, in order to do these we needed to understand the basics first. One particular type of topic we wanted to focus on was about robots avoiding obstacles in a simulation framework.

The importance to have the knowledge about these software or just have a basic understanding of how they work and what they do is high. This is because robotics is a growing field. Therefore, we think it's beneficial for us to be aware of at least one software and understand its basics. It will not only help us in the future but also give us the exposure to the practical application side of this course. The topic of robots avoiding obstacles in a simulation framework would help us to learn various aspects of a software and we all know it is very important for a robot to know its navigation properly.

The presented research describes how to implement a Robot Operating System (ROS)-based, the autonomous navigation system on a simple two wheeled mobile robot. ROS is a platform where we can create various types of ROS projects and carry out research. Over the course of this project, we have learnt many different things regarding ROS.

To use ROS, we had used an online website of RDS (ROS Development Studio) by The Construct. This website is free to an extent and we were able to learn the basic idea of how it all works.

In this experiment, we design a simple two wheeled robot and then try out obstacle avoidance algorithms so the robot avoids the obstacles in its path. We divided the overall experiment into six parts: designing the robot, organising the files, adding a laser scan sensor to the robot, writing an obstacle avoidance algorithm for which we used ROS GMapping in our 2 wheeled robot to generate a map using SLAM technique. We used the robot Laser Scan and Odometry data to generate this map.

The results we obtained was a successful simulation of the two wheeled robot avoiding obstacles in its path multiple times.

In addition to this, we have successfully understood the basics as our project was successful.

TABLE OF CONTENTS

TITLE **PAGE NO.**

1. Introduction

1.1 Problem Statement

The problem statement we worked towards is how to design a two wheeled robot and code it in a way it avoids obstacles in its path in a simulation environment.

1.2 Importance of Problem

In recent eras, the need for robotics is nothing but rising continuously. So understanding the fact that robotics is a vast field and has a variety of uses, we chose the current topic since we believe it could potentially have a lot of uses in the future and help simulating the robot in a virtual environment, give feedback and help us analyse the physical aspect of the feedback provided by the simulation. We also considered the fact that the robotics industry has a long future ahead of it since educating ourselves and giving us an exposure to the industry could significantly boost our careers in the same industry.

1.3 Aim of the Project

This project is about using an Autonomous Two Wheeled Mobile Robot to explore features and tools provided by ROS (Robot Operating System) and simulate it via ROS development studio by The Construct.

By exploring the features, we want to be able to use ROS to design a two wheeled robot which avoids obstacles via the obstacle avoidance algorithm and understand its applications. For this we used ROS GMapping in our 2 wheeled robot to generate a map using SLAM technique.We used the robot Laser Scan and Odometry data to generate the map.

1.4 Objective of the Project

1. To be able to build a robot from scratch
2. To be able to make algorithms for obstacle avoidance that will be used in the robot
3. To be able to understand and implement the codes
4. To carry out final tests and check if the project is successful

5. To make sure the project is accurate and reliable

1.5 Related Literature

For this project, the research paper we mainly focused on is **"Autonomous Navigation with Collision Avoidance using ROS"** by Journal of Remote Sensing GIS & Technology which was published on May 7, 2019. This disclosure describes the concept of Simultaneous localization and mapping (SLAM) on ROS to autonomously navigate and simultaneously avoid any collision. Furthermore, This paper focuses on RRT which is an algorithm used in autonomous robot motion planning which can easily handle problems related to obstacles by finding the probability of collision between the obstacle and itself and moves away accordingly. If there is any collision detected then RRT algorithm will create the best possible path to reach the destination and the robot will travel according to that path and reaches its destination

Another paper we focused on is **" Mobile Robot Navigation And Obstacle Avoidance Techniques "** by International Robotics & Automation Journal which was received on January 07, 2017 and published on May 23, 2017. This paper has in depth details for the different types of algorithm used in navigation and obstacle avoidance and various other soft computing techniques. From this Literature survey it is observed that many researchers have used soft computing techniques only in static environments and many of them have only done simulation work without implementing a physical robot. The various soft computing algorithms have been applied by researchers for mobile robot navigation and obstacle avoidance in different environments. Intelligent navigation techniques which are capable of navigating a mobile robot autonomously in static and dynamic environments were analysed.

International Journal of Engineering Research & Technology (IJERT) published a paper titled **"Comparison of Various Obstacle Avoidance Algorithms"** which was published on December-2015 with Vol. 4 Issue 12. This paper discussed the issue of autonomous mobile robot navigation and the analysis of different current solutions to the problem of the merits and demerits of each one. They put forward a comparison of different algorithms for preventing obstacles. Obstacle avoidance is the autonomous navigation backbone, as it helps the robot to

reach the target spot, avoiding obstacles along the way. Researchers have proposed several approaches for understanding obstacle detection and collision avoidance. Some of the commonly accepted algorithms, from the most simplistic of all 'bug algorithms' to sophisticated 'optical flow' algorithms, were discussed in this paper.

1.6 Scope of the Project

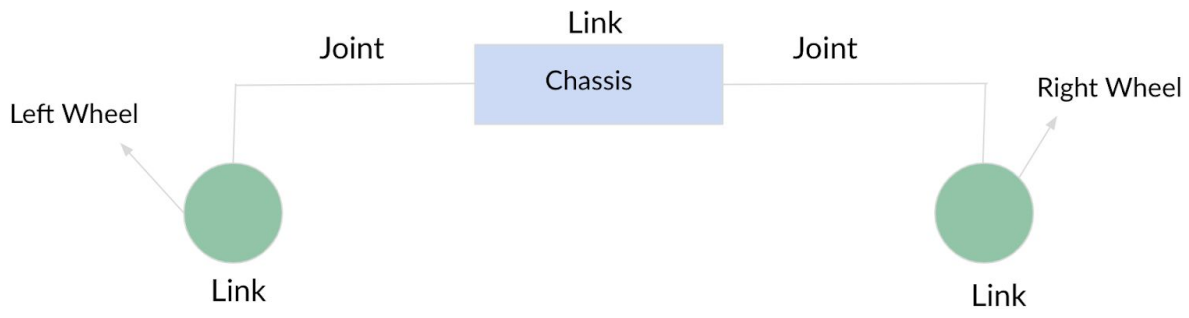
The experiment we have carried out uses a simple two wheeled robot. This experiment can be made complex using different types of robots, with different obstacles in its path. The robot chooses only one path and doesn't compare different paths and choose the best one, which we aim to implement in the same software. We test the design in a much more natural and rough environment to see if the robot can traverse along it and whether it allows us to map the terrain and explore unknown areas. The same could also be used for delivering items in places such as battlefields, containment regions or nuclear zones...etc. basically places humans cannot easily access. The proposed design is integrated with a communication system and could also help blind people move much more easier communicating and instructing them.

2. Methodology

2.1 Designing a Simple Two Wheeled Robot using URDF

In this part, we design a simple two wheeled robot using URDF files. For this, we need to have a simple block diagram that we will follow.

A simple two wheeled robot diagram:



Once we are sure of the simple diagram of the robot we want to design, we start the software work. We create a new project on the RDS website and open up the project. Then we create a workspace. For this we need to create a package so the terminal will be used. Once this is done, we create a folder and in that we create a file with the .xacro extension. Our file name is m2wr.xacro. '.xacro' is an XML macrolanguage used for URDF simplification. This is where we will code most of the designing aspect

In this file, we code by creating tags. Few tags we used were robot, link, pose, collision, inertial properties, visual etc. We code for the main link (chassis), right wheel link, left wheel link, right wheel joint and left wheel joint.

To get the standard codes for links and joints we refer to the ROS websites.

Once the design file is ready, we make a launch file called rvis.launch. This file is used to launch the rviz where we can visualise the robot we designed.

Then we use the terminal to run the ROS URDF file and launch file to view our robot on rvis. Now the designing stage is complete.

To make the robot move, we needed to use a gazebo. A new launch file was created called spawn.launch. This file is used to put the robot inside a gazebo environment. We use the terminal to run the m2wr file and then spawn file. Then we launch a the gazebo simulation and the view the log

In the xacro file, we used a gazebo plug in called differential drive controller which is made for controlling and sending messages such as angular velocities through a differential drive robot.

We use the keyboard to send messages which go through the gazebo plug in and get converted to torque for the wheels.

2.2 Organising XACRO files

In this part we focused on organising our files using two methods

- Creating another file apart from the m2wr.xacro but the other file is .xacros and the 'cat' command is used in the m2wr to read the content of the other file
- Another method is to generate tags

2.3 Inserting a laser scan sensor to the robot

Now that we have a moving robotic simulation model, we try to insert a laser scan sensory to the robot

There are two subparts:

- New element in URDF
- Gazebo plug in element

The URDF element is used because we need to design the sensor and code its parameters. The gazebo plug in element is what assigns the sensory behaviour to the sensor. The standard codes

for these were used from the ROS website as we needed the correct syntax for the codes.

Then we run and launch the files to launch the gazebo simulation and add blocks or obstacles close to the robot. Finally, we run and launch rviz to observe the sensor's detection outputs.

2.4 Reading the laser scan sensor values

The main purpose of this step was to make sure the bot reads the value from the laser scanner.

- In this step after opening up the previous work a new package is created and new libraries are added such as ospy, std_msgs, geometry_msgs and sensor_msgs and this package is named motion_plan

- Under this we create a directory in which it will contain the files(python scripts) that we will use to read the laser scan data coming on the /m2wr/laser/scan topic

- After this we spawn our bot in a gazebo simulation

- The values obtained from these functions tells us the value around 180 degrees which constitute 5 sectors (left, center-left, center, center-right, right). which has around 720 readings

2.5 Motion Planning (Bug 2 Algorithm)

Bug 2 algorithm is a modified form of Bug 1 algorithm. Instead of searching for a minimum distance point, Bug 2 algorithm focuses on maintaining direction of motion towards goal. It calculates slope of the line joining initial point and desired point. When a robot encounters an obstacle, it starts moving along the edge of the obstacle until it finds a point with the same slope. It starts moving on the line joining the point of departure and goal. This algorithm provides better performance than the Bug 1 algorithm in the majority of cases we found and tested upon.

The robot does not need to encircle the entire object as in the Bug 1 algorithm. This method can be illustrated in three basic steps:

1. Head toward goal on the m-line
2. If an obstacle is in the way, follow it until you encounter the m-line again.
3. Leave the obstacle and continue toward the goal

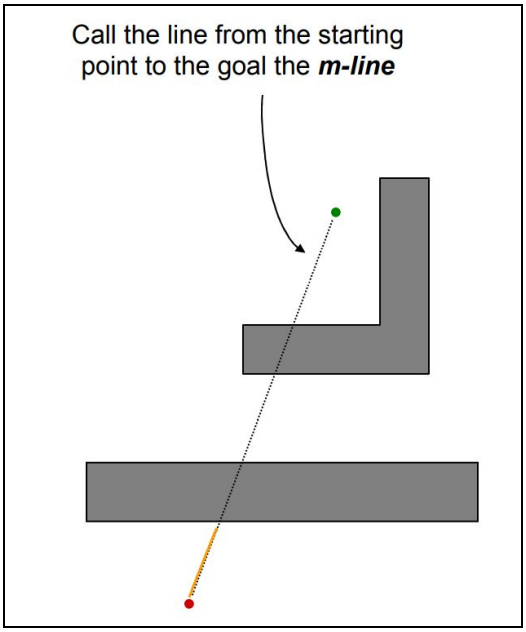


Fig 1. The bot heads towards the goal on the m-line

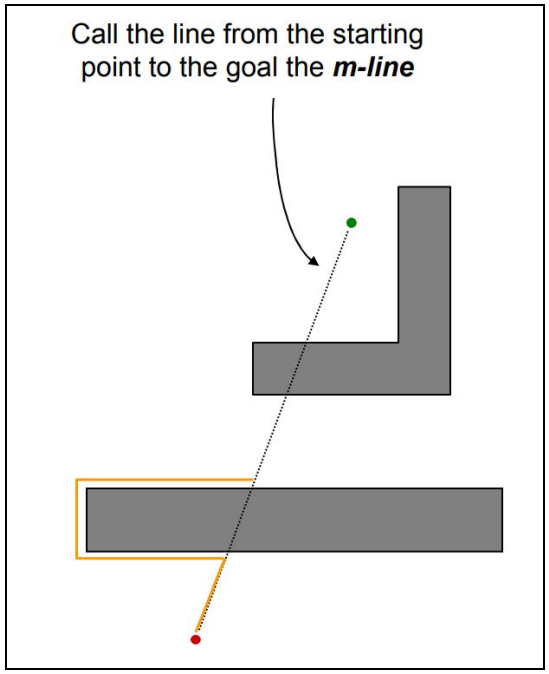


Fig 2. If an obstacle is in the way, the bot follows it until it encounters the m-line again

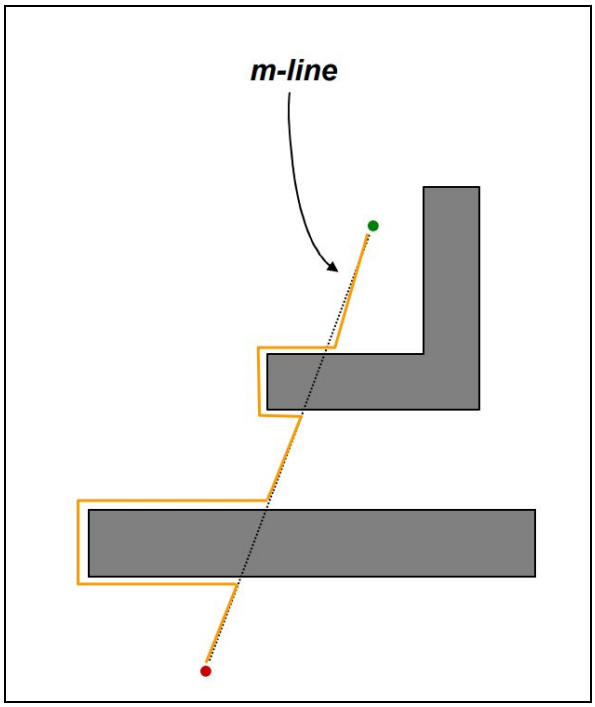
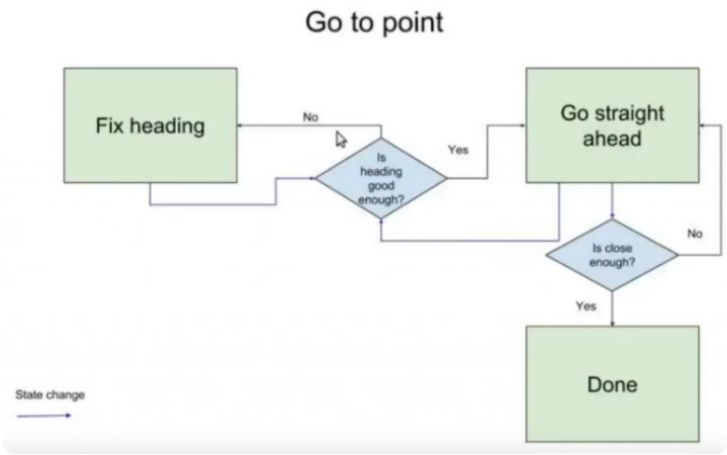


Fig 3. The bot leaves the obstacle and continues towards the goal

For this the important functions used in the bug2.py script are:

- **main():** This is the function through which the program starts running. It initialises a node, two subscribers namely laser scan and odometry and two service clients which is the go_to_point_switch and the wall_follower_switch. A state based logic is used to drive the 2 wheeled robot towards the goal position. Initially the robot is put in Go to point state, and when an obstacle is detected the state is switched to Wall following state. When there is a straight free path towards the goal the robot again switches back to state Go to point.
- **change_state()**: This function accepts a state argument and calls the respective service handler. In our case, we use two states, which are:
    1. State 1 : **Go to point:** This is a simple navigation algorithm to move our robot from any point to a desired point. We used the concept of state machines to implement the navigation logic. The robot can be in any one state at a time and can switch to other states as different conditions arise. This is depicted by the following state transition diagram



    2. State 2 : **Wall following:** This algorithm involves primarily three functions:
        - ❖ find_wall : This function defines the action to be taken by the robot when it is not surrounded by any obstacle. This method essentially makes the robot move in an anti-clockwise circle (until it finds a wall).

- ❖ turn_left : This function executes the turn left action when the robot detects an obstacle.
- ❖ follow_the_wall : Once the robot is positioned such that its front and front-left path is clear while its front-right is obstructed the robot goes into the follow wall state and hence this function makes the robot follow a straight line.

This is the overall logic that governs the wall following behavior of the robot.

- **distance_to_line():** This function calculates the distance of the robot from the imaginary_line that joins the initial position of the robot with the desired position of the robot. We also make use of the count_state_time_ variable to track time elapsed since changing state. This helps us to wrongly trigger state transition on the first contact with the imaginary line on the first encounter. We let some time go by before we seek to find the imaginary line after we have started circumnavigating the obstacle. We use the following formula for distance_to_line function:

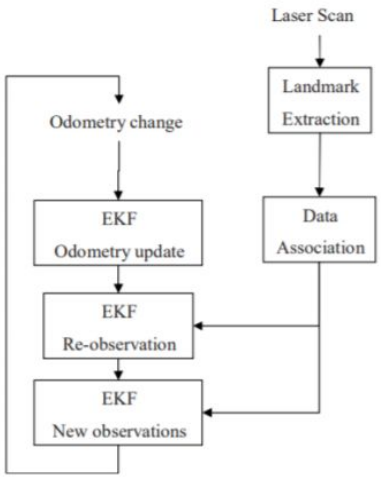
If the line passes through two points $P_1=(x_1,y_1)$ and $P_2=(x_2,y_2)$ then the distance of (x_0,y_0) from the line is:

$$\text{distance}(P_1, P_2, (x_0, y_0)) = \frac{|(y_2 - y_1)x_0 - (x_2 - x_1)y_0 + x_2y_1 - y_2x_1|}{\sqrt{(y_2 - y_1)^2 + (x_2 - x_1)^2}}.$$

- **clbk_odom():** This function receives the odometry data and extracts the position and yaw information.
- **clbk_laser():** This function is passed to the Subscriber method and it executes when a new laser data is made available. This function writes distance values in the global variable regions_ and calls the function take_actions.

2.6 GMapping and SLAM

SLAM is used so that the robot knows what all is there in the environment and compares its position with respect to the environment whereas gmapping is used for implementing SLAM as it acts the laser scanner. The success of SLAM is determined by the fact that the more the robot explores the environments the better will be the quality of the map and its current position. SLAM has different stages which include identifying location of landmarks, data associate, state estimate, state update and landmark update. Using this information 2D and 3D movements are made possible.



This overview of how the SLAM process is used to map the environment around it.

We use the gmapping package in our simulation in order to implement SLAM. In this we initialise different parameters such as:

- Scan_topic: This helps read the values from the laser scan
- Base_frame: indicates the robot base element frame which in our case is the main chassis
- Odom_frame: Helps us get the odometry data

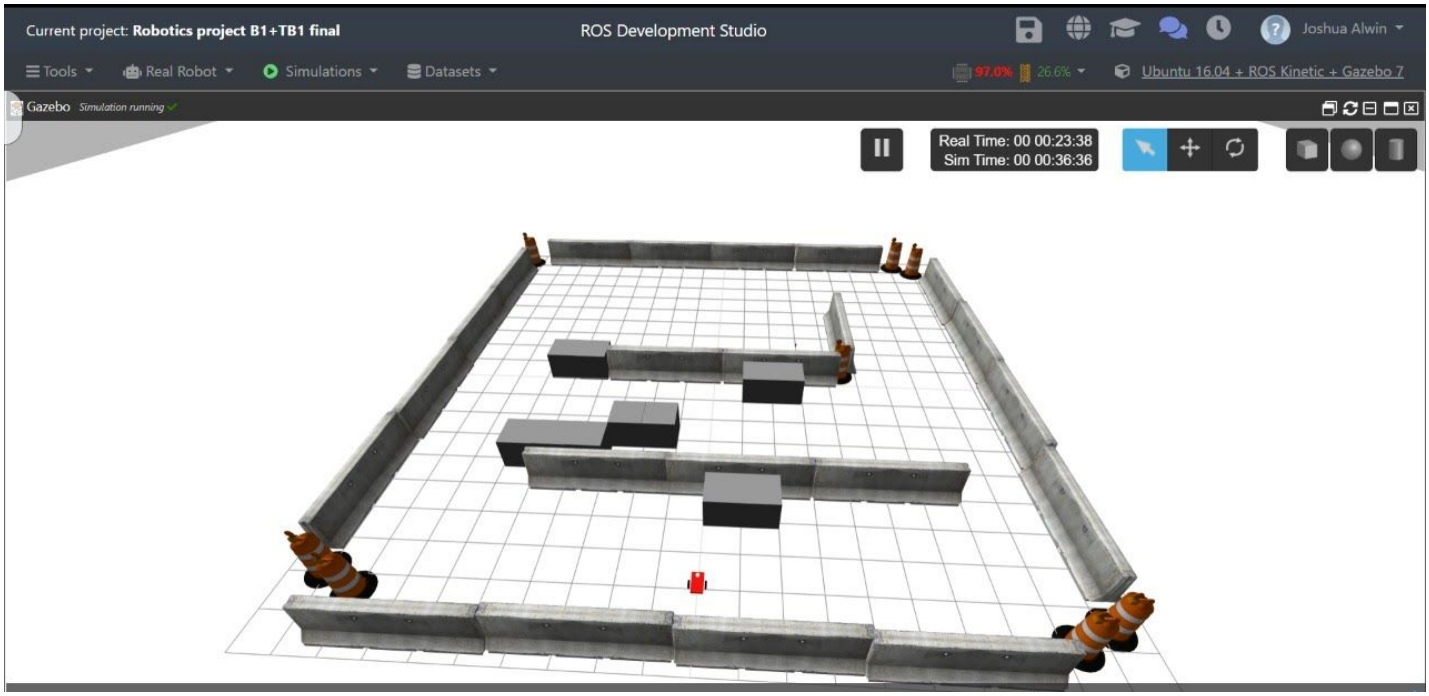
And the packages we used to implement this are:

- Robot_state_publisher
- Rviz
- Gmapping

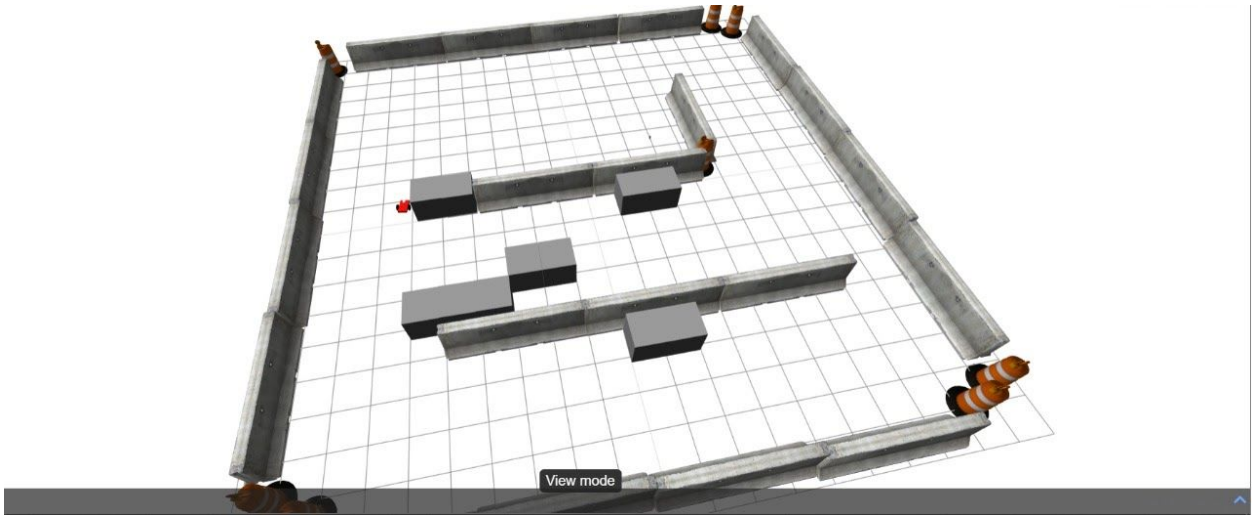
3. Results and Discussion

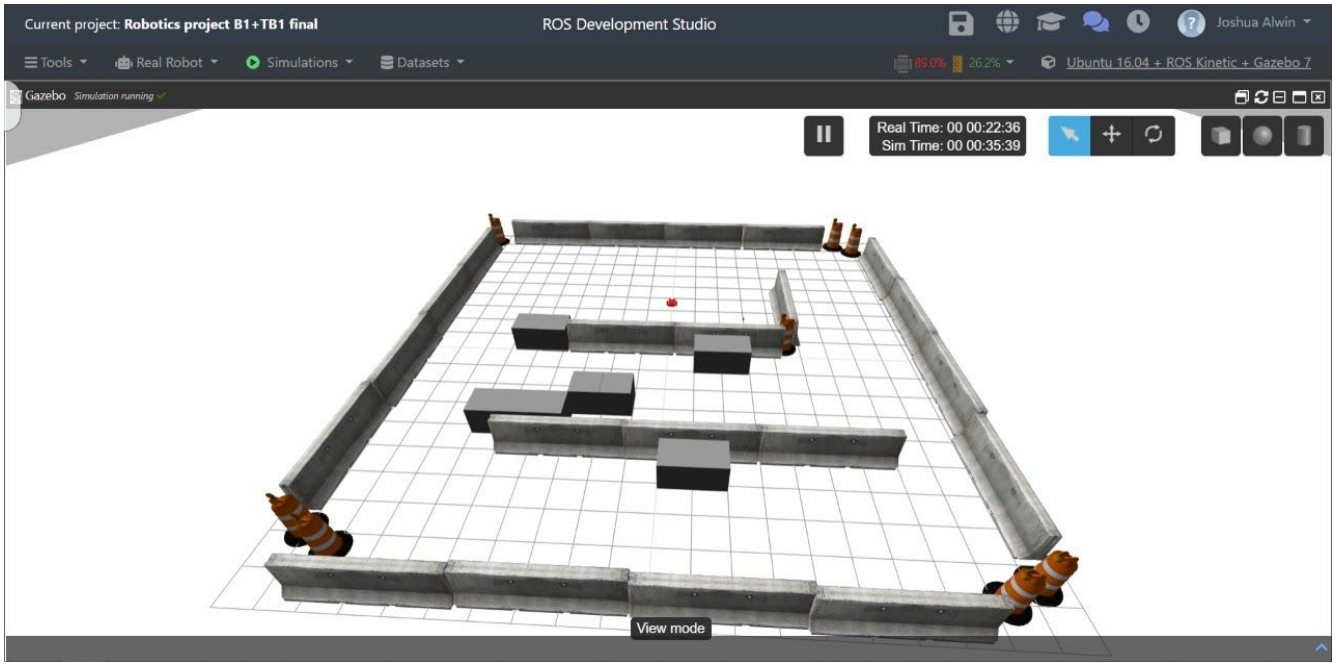
A simulation of a robot in RDS website was successfully made along with a map in which the robot's ability to avoid obstacles was proved using the BUG 2 algorithm which uses GMapping and SLAM to traverse along the created map. The results that are obtained by simulation and the output of the gmap is shown below:

Starting position of the robot:

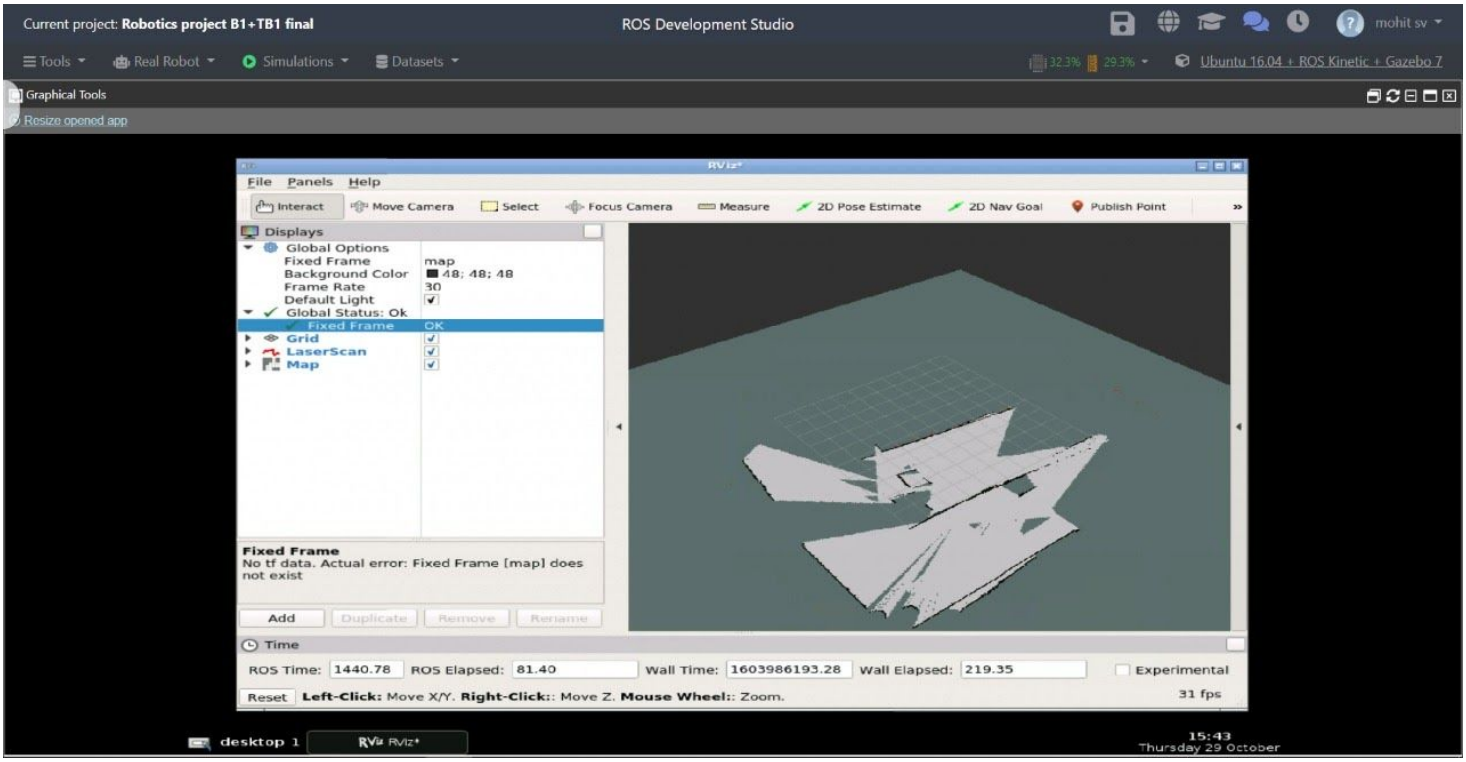


Continuation and final:


View mode


Current project: Robotics project B1+TB1 final
ROS Development Studio
Joshua Alwin
Tools
Real Robot
Simulations
Datasets
Ubuntu 16.04 + ROS Kinetic + Gazebo 7
Gazebo
Simulation running
Real Time: 00 00:22:36
Sim Time: 00 00:35:39
View mode

Gmap:



4. Conclusion and Recommendations

The major results obtained by this experiment are that we were able to design a two wheeled robot using the RDS website. After this, we constructed an autonomous robot and coded for obstacle avoidance. The result we achieved from this was a successful attempt for the robot to avoid obstacles in its path multiple times.

We have fulfilled the objectives we set at the beginning of the experiment. The main objective being having a basic understanding and making sure the project is reliable and accurate. The recommendations we would provide to anyone carrying out a similar project to ours would be to improve the algorithm being used. A better way to plan the algorithm would be to include parameters such that each time the robot simulation is carried out, the robot chooses the best possible path for itself while avoiding the obstacles.

5. Appendices

5.1 Code for robot designing

Defining the chassis:

m2wr.xacro × macros.xacro × materials.xacro

```
<?xml version="1.0" ?>
<robot name="m2wr" xmlns:xacro="http://www.ros.org/wiki/xacro">

  <xacro:include filename="$(find m2wr_description)/urdf/materials.xacro" />
  <xacro:include filename="$(find m2wr_description)/urdf/m2wr.gazebo" />
  <xacro:include filename="$(find m2wr_description)/urdf/macros.xacro" />

  <link name="link_chassis">
    <!-- pose and inertial -->
    <pose>0 0 0.1 0 0 0</pose>
    <inertial>
      <mass value="5"/>
      <origin rpy="0 0 0" xyz="0 0 0.1"/>
      <inertia ixx="0.0395416666667" ixy="0" ixz="0" iyy="0.106208333333" iyz="0" izz="0.106208333333"/>
    </inertial>
    <!-- body -->
    <collision name="collision_chassis">
      <geometry>
        <box size="0.5 0.3 0.07"/>
      </geometry>
    </collision>
    <visual>
      <origin rpy="0 0 0" xyz="0 0 0"/>
      <geometry>
        <box size="0.5 0.3 0.07"/>
      </geometry>
      <material name="blue"/>
    </visual>
    <!-- caster front -->
    <collision name="caster_front_collision">
      <origin rpy=" 0 0 0" xyz="0.35 0 -0.05"/>
      <geometry>
        <sphere radius="0.05"/>
      </geometry>
      <surface>
        <friction>
          <ode>
            <mu>0</mu>
            <mu2>0</mu2>
            <slip1>1.0</slip1>
            <slip2>1.0</slip2>
          </ode>
        </friction>
      </surface>
    </collision>
    <visual name="caster_front_visual">
      <origin rpy=" 0 0 0" xyz="0.2 0 -0.05"/>
      <geometry>
        <sphere radius="0.05"/>
```

m2wr.xacro × macros.xacro materials.xacro

```
            <ode>
              <mu>0</mu>
              <mu2>0</mu2>
              <slip1>1.0</slip1>
              <slip2>1.0</slip2>
            </ode>
          </friction>
        </surface>
      </collision>
      <visual name="caster_front_visual">
        <origin rpy=" 0 0 0" xyz="0.2 0 -0.05"/>
        <geometry>
          <sphere radius="0.05"/>
        </geometry>
      </visual>
    </link>
      <visual>
        <origin xyz="0 0 0" rpy="0 0 0" />
        <geometry>
          <cylinder radius="0.05" length="0.1"/>
        </geometry>
        <material name="white" />
      </visual>

      <collision>
        <origin xyz="0 0 0" rpy="0 0 0"/>
        <geometry>
          <cylinder radius="0.05" length="0.1"/>
        </geometry>
      </collision>
    </link>


    <xacro:link_wheel name="link_right_wheel" />
    <xacro:joint_wheel name="joint_right_wheel" child="link_right_wheel" origin_xyz="-0.05 0.20 0" />

    <xacro:link_wheel name="link_left_wheel" />
    <xacro:joint_wheel name="joint_left_wheel" child="link_left_wheel" origin_xyz="-0.05 -0.20 0" />
  </robot>
```

Defining Links and Joints:

m2wr.xacro | macros.xacro | materials.xacro

```
<?xml version="1.0"?>
<robot xmlns:xacro="http://www.ros.org/wiki/xacro">
    <xacro:macro name="link_wheel" params="name">
        <link name="${name}">
            <inertial>
              <mass value="0.2"/>
              <origin rpy="0 1.5707 1.5707" xyz="0 0 0"/>
              <inertia ixx="0.000526666666667" ixy="0" ixz="0" iyy="0.000526666666667" iyz="0" izz="0.001"/>
            </inertial>
            <collision name="link_right_wheel_collision">
              <origin rpy="0 1.5707 1.5707" xyz="0 0 0"/>
              <geometry>
                <cylinder length="0.04" radius="0.1"/>
              </geometry>
            </collision>
            <visual name="${name}_visual">
              <origin rpy="0 1.5707 1.5707" xyz="0 0 0"/>
              <geometry>
                <cylinder length="0.04" radius="0.1"/>
              </geometry>
            </visual>
        </link>
    </xacro:macro>

    <xacro:macro name="joint_wheel" params="name child origin_xyz">
      <joint name="${name}" type="continuous">
        <origin rpy="0 0 0" xyz="${origin_xyz}"/>
        <child link="${child}"/>
        <parent link="link_chassis"/>
        <axis rpy="0 0 0" xyz="0 1 0"/>
        <limit effort="10000" velocity="1000"/>
        <joint_properties damping="1.0" friction="1.0"/>
      </joint>
    </xacro:macro>

    <xacro:macro name="cylinder_inertia" params="mass r l">
      <inertia  ixx="${mass*(3*r*r+l*l)/12}" ixy = "0" ixz = "0"
                iyy="${mass*(3*r*r+l*l)/12}" iyz = "0"
                izz="${mass*(r*r)/2}" />
    </xacro:macro>
</robot>
```

Defining materials:

m2wr.xacro | macros.xacro | materials.xacro ×

```
<?xml version="1.0"?>
<robot xmlns:xacro="http://www.ros.org/wiki/xacro">
  <material name="black">
    <color rgba="0.0 0.0 0.0 1.0"/>
  </material>
  <material name="blue">
    <color rgba="0.203125 0.23828125 0.28515625 1.0"/>
  </material>
  <material name="green">
    <color rgba="0.0 0.8 0.0 1.0"/>
  </material>
  <material name="grey">
    <color rgba="0.2 0.2 0.2 1.0"/>
  </material>
  <material name="orange">
    <color rgba="1.0 0.423529411765 0.0392156862745 1.0"/>
  </material>
  <material name="brown">
    <color rgba="0.870588235294 0.811764705882 0.764705882353 1.0"/>
  </material>
  <material name="red">
    <color rgba="0.80078125 0.12890625 0.1328125 1.0"/>
  </material>
  <material name="white">
    <color rgba="1.0 1.0 1.0 1.0"/>
  </material>
</robot>
```

5.2 Code for laser scan sensor to insert in robot

m2wr.gazebo × | reading_laser.py

```
<?xml version="1.0"?>
<robot xmlns:xacro="http://www.ros.org/wiki/xacro">

  <gazebo reference="link_chassis">
    <material>Gazebo/Red</material>
  </gazebo>
  <gazebo reference="link_left_wheel">
    <material>Gazebo/Black</material>
  </gazebo>
  <gazebo reference="link_right_wheel">
    <material>Gazebo/Black</material>
  </gazebo>

  <gazebo>
    <plugin filename="libgazebo_ros_diff_drive.so" name="differential_drive_controller">
      <alwaysOn>true</alwaysOn>
      <updateRate>20</updateRate>
      <leftJoint>joint_left_wheel</leftJoint>
      <rightJoint>joint_right_wheel</rightJoint>
      <wheelSeparation>0.4</wheelSeparation>
      <wheelDiameter>0.2</wheelDiameter>
      <torque>0.1</torque>
      <commandTopic>cmd_vel</commandTopic>
      <odometryTopic>odom</odometryTopic>
      <odometryFrame>odom</odometryFrame>
      <robotBaseFrame>link_chassis</robotBaseFrame>
    </plugin>
  </gazebo>
```

m2wr.gazebo × reading_laser.py

```
      <wheelSeparation>0.4</wheelSeparation>
      <wheelDiameter>0.2</wheelDiameter>
      <torque>0.1</torque>
      <commandTopic>cmd_vel</commandTopic>
      <odometryTopic>odom</odometryTopic>
      <odometryFrame>odom</odometryFrame>
      <robotBaseFrame>link_chassis</robotBaseFrame>
    </plugin>
  </gazebo>

  <gazebo reference="sensor_laser">
    <sensor type="ray" name="head_hokuyo_sensor">
      <pose>0 0 0 0 0 0</pose>
      <visualize>false</visualize>
      <update_rate>20</update_rate>
      <ray>
        <scan>
          <horizontal>
            <samples>720</samples>
            <resolution>1</resolution>
            <min_angle>-1.570796</min_angle>
            <max_angle>1.570796</max_angle>
          </horizontal>
        </scan>
        <range>
          <min>0.10</min>
          <max>10.0</max>
          <resolution>0.01</resolution>
        </range>
        <noise>
          <type>gaussian</type>
          <mean>0.0</mean>
          <stddev>0.01</stddev>
        </noise>
      </ray>
      <plugin name="gazebo_ros_head_hokuyo_controller" filename="libgazebo_ros_laser.so">
        <topicName>/m2wr/laser/scan</topicName>
        <frameName>sensor_laser</frameName>
      </plugin>
    </sensor>
  </gazebo>

</robot>
```

5.3 Code for reading laser scan sensor values

reading_laser.py

```python
#! /usr/bin/env python

import rospy

from sensor_msgs.msg import LaserScan

def clbk_laser(msg):
    # 720 / 5 = 144
    regions = [
        min(min(msg.ranges[0:143]), 10),
        min(min(msg.ranges[144:287]), 10),
        min(min(msg.ranges[288:431]), 10),
        min(min(msg.ranges[432:575]), 10),
        min(min(msg.ranges[576:713]), 10),
    ]
    rospy.loginfo(regions)

def main():
    rospy.init_node('reading_laser')

    sub = rospy.Subscriber('/m2wr/laser/scan', LaserScan, clbk_laser)

    rospy.spin()

if __name__ == '__main__':
    main()

```

5.4 Code for Motion Planning (Bug 2 Algorithm)

bug2.py

```python
#! /usr/bin/env python

# import ros stuff
import rospy
# import ros message
from geometry_msgs.msg import Point
from sensor_msgs.msg import LaserScan
from nav_msgs.msg import Odometry
from tf import transformations
from gazebo_msgs.msg import ModelState
from gazebo_msgs.srv import SetModelState
# import ros service
from std_srvs.srv import *

import math

srv_client_go_to_point_ = None
srv_client_wall_follower_ = None
yaw_ = 0
yaw_error_allowed_ = 5 * (math.pi / 180) # 5 degrees
position_ = Point()
initial_position_ = Point()
initial_position_.x = rospy.get_param('initial_x')
initial_position_.y = rospy.get_param('initial_y')
initial_position_.z = 0
desired_position_ = Point()
desired_position_.x = rospy.get_param('des_pos_x')
desired_position_.y = rospy.get_param('des_pos_y')
desired_position_.z = 0
regions_ = None
state_desc_ = ['Go to point', 'wall following']
state_ = 0
count_state_time_ = 0 # seconds the robot is in a state
count_loop_ = 0
# 0 - go to point
# 1 - wall following

# callbacks
def clbk_odom(msg):
    global position_, yaw_

    # position
    position_ = msg.pose.pose.position

    # yaw
    quaternion = (
        msg.pose.pose.orientation.x,
        msg.pose.pose.orientation.y,
        msg.pose.pose.orientation.z,
```

bug2.py

```
        msg.pose.pose.orientation.x,
        msg.pose.pose.orientation.y,
        msg.pose.pose.orientation.z,
        msg.pose.pose.orientation.w)
    euler = transformations.euler_from_quaternion(quaternion)
    yaw_ = euler[2]

def clbk_laser(msg):
    global regions_
    regions_ = {
        'right':  min(min(msg.ranges[0:143]), 10),
        'fright': min(min(msg.ranges[144:287]), 10),
        'front':  min(min(msg.ranges[288:431]), 10),
        'fleft':  min(min(msg.ranges[432:575]), 10),
        'left':   min(min(msg.ranges[576:719]), 10),
    }

def change_state(state):
    global state_, state_desc_
    global srv_client_wall_follower_, srv_client_go_to_point_
    global count_state_time_
    count_state_time_ = 0
    state_ = state
    log = "state changed: %s" % state_desc_[state]
    rospy.loginfo(log)
    if state_ == 0:
        resp = srv_client_go_to_point_(True)
        resp = srv_client_wall_follower_(False)
    if state_ == 1:
        resp = srv_client_go_to_point_(False)
        resp = srv_client_wall_follower_(True)

def distance_to_line(p0):
    # p0 is the current position
    # p1 and p2 points define the line
    global initial_position_, desired_position_
    p1 = initial_position_
    p2 = desired_position_
    # here goes the equation
    up_eq = math.fabs((p2.y - p1.y) * p0.x - (p2.x - p1.x) * p0.y + (p2.x * p1.y) - (p2.y * p1.x))
    lo_eq = math.sqrt(pow(p2.y - p1.y, 2) + pow(p2.x - p1.x, 2))
    distance = up_eq / lo_eq

    return distance


def normalize_angle(angle):
    if(math.fabs(angle) > math.pi):
        angle = angle - (2 * math.pi * angle) / (math.fabs(angle))
```

bug2.py

```python

def normalize_angle(angle):
    if(math.fabs(angle) > math.pi):
        angle = angle - (2 * math.pi * angle) / (math.fabs(angle))
    return angle

def main():
    global regions_, position_, desired_position_, state_, yaw_, yaw_error_allowed_
    global srv_client_go_to_point_, srv_client_wall_follower_
    global count_state_time_, count_loop_

    rospy.init_node('bug0')

    sub_laser = rospy.Subscriber('/m2wr/laser/scan', LaserScan, clbk_laser)
    sub_odom = rospy.Subscriber('/odom', Odometry, clbk_odom)

    rospy.wait_for_service('/go_to_point_switch')
    rospy.wait_for_service('/wall_follower_switch')
    rospy.wait_for_service('/gazebo/set_model_state')

    srv_client_go_to_point_ = rospy.ServiceProxy('/go_to_point_switch', SetBool)
    srv_client_wall_follower_ = rospy.ServiceProxy('/wall_follower_switch', SetBool)
    srv_client_set_model_state = rospy.ServiceProxy('/gazebo/set_model_state', SetModelState)

    # set robot position
    model_state = ModelState()
    model_state.model_name = 'm2wr'
    model_state.pose.position.x = initial_position_.x
    model_state.pose.position.y = initial_position_.y
    resp = srv_client_set_model_state(model_state)

    # initialize going to the point
    change_state(0)

    rate = rospy.Rate(20)
    while not rospy.is_shutdown():
        if regions_ == None:
            continue

        distance_position_to_line = distance_to_line(position_)

        if state_ == 0:
            if regions_['front'] > 0.15 and regions_['front'] < 1:
                change_state(1)

        elif state_ == 1:
            if count_state_time_ > 5 and \
               distance_position_to_line < 0.1:
                change_state(0)

        count_loop_ = count_loop_ + 1
        if count_loop_ == 20:
            count_state_time_ = count_state_time_ + 1
            count_loop_ = 0

        rospy.loginfo("distance to line: [%.2f], position: [%.2f, %.2f]", distance_to_line(position_), position_.x, position_.y)
        rate.sleep()

if __name__ == "__main__":
    main()
```

5.5 Code for GMapping and SLAM

gmapping.launch

```
<launch>
  <arg name="scan_topic"  default="/m2wr/laser/scan" />
  <arg name="base_frame"  default="link_chassis"/>
  <arg name="odom_frame"  default="odom"/>              <!--Odometry framce-->

  <node pkg="robot_state_publisher" type="robot_state_publisher" name="robot_state_publisher"></node>

  <node pkg="rviz" type="rviz" name="rviz"></node>

  <node pkg="gmapping" type="slam_gmapping" name="slam_gmapping" output="screen">
    <param name="base_frame" value="$(arg base_frame)"/>
    <param name="odom_frame" value="$(arg odom_frame)"/>
    <param name="map_update_interval" value="5.0"/>
    <param name="maxUrange" value="6.0"/>
    <param name="maxRange" value="8.0"/>
    <param name="sigma" value="0.05"/>
    <param name="kernelSize" value="1"/>
    <param name="lstep" value="0.05"/>
    <param name="astep" value="0.05"/>
    <param name="iterations" value="5"/>
    <param name="lsigma" value="0.075"/>
    <param name="ogain" value="3.0"/>
    <param name="lskip" value="0"/>
    <param name="minimumScore" value="200"/>
    <param name="srr" value="0.01"/>
    <param name="srt" value="0.02"/>
    <param name="str" value="0.01"/>
    <param name="stt" value="0.02"/>
    <param name="linearUpdate" value="0.5"/>
    <param name="angularUpdate" value="0.436"/>
    <param name="temporalUpdate" value="-1.0"/>
    <param name="resampleThreshold" value="0.5"/>
    <param name="particles" value="80"/>

    <param name="xmin" value="-1.0"/>
    <param name="ymin" value="-1.0"/>
    <param name="xmax" value="1.0"/>
    <param name="ymax" value="1.0"/>

    <param name="delta" value="0.05"/>
    <param name="llsamplerange" value="0.01"/>
    <param name="llsamplestep" value="0.01"/>
    <param name="lasamplerange" value="0.005"/>
    <param name="lasamplestep" value="0.005"/>
    <remap from="scan" to="$(arg scan_topic)"/>
  </node>
</launch>
```

6. References

1. Akash, Ananya Naik, Sandeep, Deepti Raj, *Autonomous Navigation with Collision Avoidance using ROS,* Journal of Remote Sensing GIS & Technology, 2019, Volume 5 Issue 2
2. Anish Pandey, Shalini Pandey, Parhi DR, *Mobile robot navigation and obstacle avoidance techniques,* International Robotics & Automation Journal, Volume 2 Issue 3 - 2017
3. Vayeda Anshav Bhavesh, *Comparison of Various Obstacle Avoidance Algorithms,* International Journal of Engineering Research & Technology (IJERT), Vol. 4 Issue 12, December-2015
4. Marco Arruda, *Exploring ROS using a 2 Wheeled Robot,* ROS Tutorials
5. S. Riisgaard, M. Rufus Blas, *SLAM for Dummies, A Tutorial Approach to Simultaneous Localization and Mapping, Massachusetts Institute of Technology*
6. *Distance from a point to a line*, Wikipedia