Technical Survey Report

Mahmoud Badran and Muhammad Arsalan Khawaja

November 14, 2020

Abstract

Navigation of a robot is a technically painstaking and complicated task. It involves a lot of skills and knowledge in different disciplines to understand and implement on Robotics. Robotics is multidisciplinary field and it naturally offers technical challenges and great insight on integration of these fields, which we intend to learn and explore. This report presents a technical survey of the robotics project for the navigation of Robot. Robotic Operating Software(ROS) is used in this project. ROS provides an environment which facilitates robotic task with the help of its powerful libraries and makes complicated robotics tasks manageable. The report starts with brief introduction of ROS. It then continues to discuss the tasks, their implementation supported by necessary visuals and data. The report concludes that there is good confidence supported with evidence that the project will be successfully completed. The code and brief guide (readme) is attached on github

Contents

Chapter 1

Introduction

Robotics Operating System (ROS), is a middle ware, low level framework, to write robotic software. It can be considered as an API to make the process of developing a robotic related projects more flexible, and simplified. There will be no need for an extensive knowledge of the hardware in which it saves much effort and time in the development phase. It includes many libraries and tools which connects and control the robot manipulators, handle the communication between multiple devices in a a working space.
 ROS is supported by many operating systems like Ubuntu, windows. Ubuntu is the more stable operating system working with ROS. However, for the development of this project we are using the construct web platform, which is an online robotics working environment. The platform uses Ubuntu as the main operating system with ROS kinetic and uses the **Gazebo** as real world simulator with many robot model simulations like TurtleBot 3. The platform will enable us to learn some of ROS basic techniques to be able to apply Robot Control, Mapping, Robot Localization, Path planning and setting up some goals to navigate through the environment.

1.1 Project Objectives

The project goal is to apply the learned ROS techniques and packages to apply the navigation task on Turtlebot3:

- Moving the robot around the environment using /**cmd_vel topic**.
- Construct a map of the whole environment. We need to fully occupy the whole environment, then we need to localize the Robot.
- Path planning, we need to publish a goal to move base navigation system in which Turtlebot3 can reach that goal without colliding with any obstacles.
- Create a way points that allows the Turtlebot3 to navigate within the environment.

1.2 Project Management

Project management is an important aspect of every project. We utilized the resources we had and faced the pressure of constraints including time. This section sheds some light on the management and problems we are facing.

1.2.1 Resources

The Construct[1] is a amazing platform to learn ROS. We took 3 courses on the construct and also tried to explore other related courses. Due to lack of computational power, unavailability of Robots due to COVID-19, we were unable to learn on real robots. We also lacked the resources to buy a TurtleBot for our learning at home. However we had staisfied experience with online simulated platform. We also found the book [2] very resourceful and guiding.

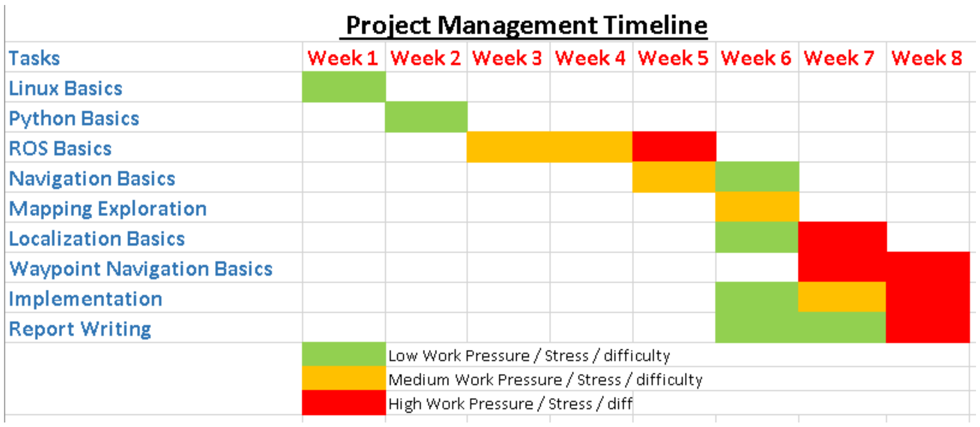
1.2.1.1 Github Repository

Since the construct was having some problems with saving the progress, we use Github to manage and write technical review. Due to confinement, Its not appreciable to work toghether everytime, Github provides us great solution of working distantly through its fundamental facility of version control. Meetings were only hold to discuss critical tasks.

The github repository can be found by clicking here. An extensive readme file guides through the tutorials for achieving the project objectives.

1.2.2 Timeline

The timeline for our project is described in figure 1.1. Alot of time was spend getting used to ROS basics and learning to be comfortable with the environment.



Project Management Timeline

Tasks	Week 1	Week 2	Week 3	Week 4	Week 5	Week 6	Week 7	Week 8
Linux Basics	🟩							
Python Basics		🟩						
ROS Basics			🟧	🟧	🟥			
Navigation Basics					🟧	🟩		
Mapping Exploration						🟧		
Localization Basics						🟩	🟥	
Waypoint Navigation Basics							🟥	🟥
Implementation						🟩	🟧	🟥
Report Writing						🟩	🟩	🟥
	🟩	Low Work Pressure / Stress / difficulty						
	🟧	Medium Work Pressure / Stress / difficulty						
	🟥	High Work Pressure / Stress / diff						

Figure 1.1: Technical Survey Management timeline created on Excel.

1.2.3 Challenges

We felt alot of pressure for our schedules and the constraint of timing was felt gravely. We also faced problems with the stability of the construct platform. A number of times it was not functional. We also lack the basic privelage of good internet connection in Residence Acacias, which makes our learning very difficult and frustrating. We tried to learn and work with our SIM card limited internet but discovered that The Construct consumes a lot data and makes it hard for us to use Mobile SIM. We basically worked mostly in university and even in lockdown, we were given the privelage of using university lab, but the working hours in university are limited. However, we have complained to our Co-ordinator David Fofi about the problem and we hope the problem will get fixed soon.

Chapter 2

Analysis and Implementation

This chapter provides the neccessary background knowledge and establishes the pre-requisites to carry out the implementation of the project. It then carves out blueprint of the implementation to be carried out for successfuly completing this project.

The important concepts to be studied and implemented here are as follows:

- **Nodes** is one of the most important concepts in ROS. We can think of node as a function. It takes some input, performs soem operation on it and gives an output. A node can subscribe or publish to a topic. Nodes are fundamental in ROS for communiation with robot and for performing a task. Nodes are written in python language for this project. The figure 2.1 demonstrates the concepts.

- **Topic** can be thought of as the wire between two nodes that transports data. Topics are the process of transmitting data between nodes. Some nodes are responsible for publishing some data to a specific topic where other nodes (subscribers) will be able to request that data (messages) from the topic.

- **Messages** are data structures that describe the data, ROS nodes publish or recieve. the nodes communicate, send messages, receive messages through topics.

- **Services** is another way to transmit the data between nodes. it is a synchronous operation where the robot has to stop processing while waiting for a service response.

The main asset of ROS is its libraries and packages.ROS package is defined as follows:

> A package in ROS is a working directory which contains all the necessary ROS files like executable python and cpp files, configuration and compilation files and launch files.

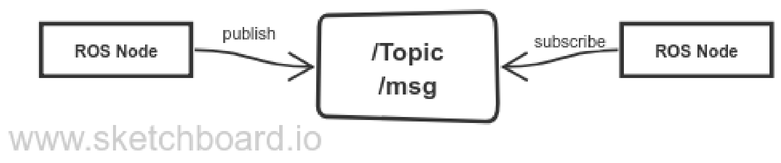


Figure 2.1: A demonstration of the node and Topic concept drawn on sketchboard. Here, the left node transmits message(data) which is called publishing. The right node recieves the data as its input which is called subscribing.

Here it is important to understand the structure of ROS package. It contains following folders:

- Source folder (**src**) which contains all the executable files.
- Launch folder (**launch**) which contains all the launch files.
- Package file (**package.xml**) which contains other packages dependencies and url paths. In case one wants to be able to call other packages from this package.
- **CMakeLists.txt** file which contains some cpp compilations and building commands.

Another important concpept in ROS in build system. The Construct in ROS uses Catkin build system. Catkin provides a smooth environment where one can code in python files and ROS will automatically take care of the implementation on hardware [**?**]. It is important to note that Catkin also provides worksapce and has many packages related to implementing specfic tasks. The working space we work on The Construct is **catkin_ws.** It is the directory where all the packages are installed. The following commands/ concepts are fundamental to carry out this project:

- We can move to any package using **roscd**

  ```
  roscd <package_name>
  ```

- ROS programs(nodes) can be executed using launch files. We can also run multiple nodes in one file. A launch file can be run using **roslaunch** command:

  ```
  roslaunch <package_name> <launch_file>
  ```

- If we want to create a package in the working space catkin_ws, we use this command:

```
catkin_create_pkg <package_name> <package_dependecies>
```

- If we want to know some information about a specific node, we use this **rosnode** command:

  ```
  rosnode info <node_name>
  ```

- If we want to know some information about a specific topic, we use this **rostopic** command:

  ```
  rostopic info <topic_name>
  ```

- If we want to know some information about a specific message, we use this **rosmsg** command:

  ```
  rosmsg show info <msg_name>
  ```

- An important command needed to be running before start working with any ROS project called **roscore**. It contains all the necessary files for any ROS system.

2.1 Implementation

The implementation plan is established in this section for the four project objectives mentioned in section 1.1.

2.1.1 Task 1: Robot Control

We use TurtleBot3, which is a mobile robot with small size and low price but still have the same quality that other mobile robots have. The course on ROS basics [3] guided us.The step by step guide is mentioned below

1. First of all, it is neccessary to **start and launch our turtlebot3**. It is done using the turtlebot3_bringup package and starting turtlebot3_remote.launch. file. The code is as follows:

   ```
   <include file="$(find turtlebot3_bringup)
                   /launch/turtlebot3_remote.launch" />
   ```

2. It is important to know about the topics. One important command we can use to know about the topics that got published by the environment is as follows:

   ```
   rostopic list
   ```

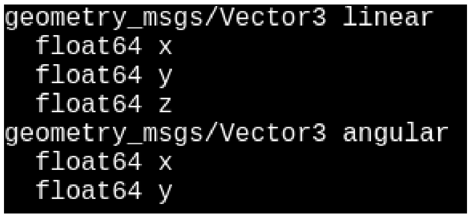
   One of the topics that should be provided to move the robot is /**cmd_vel**:

3. With this topic, we can publish velocity information to the robot. if we want to know more information about this topic we can execute this command:

   ```
   rostopic info /cmd_vel
   ```

4. After running the command we can see that this topic uses Twist type messages. It can be observed in figure 2.2 that this topic recieves data of type Twist (angular and linear velocities ,(x,y,z)). To know more information about the message we execute this command:

   ```
   rosmsg show geometry_msgs/Twist
   ```



```
geometry_msgs/Vector3 linear
  float64 x
  float64 y
  float64 z
geometry_msgs/Vector3 angular
  float64 x
  float64 y
```

Figure 2.2: The message contains three linear translations in x,y,z directions. The bottom two show the two rotations of robot. These five quantities are important to know for the sake of controlling the robot.

   TurtleBot3 receives the velocity information by subscribing to this topic. The topic will provide the robot translation and rotation data.

5. We can publish to this topic by running this command:

   ```
   rostopic pub /cmd_vel TAB TAB
   ```

   And then in the terminal we can edit the values of the two vectors.

6. Otherwise, we can create a launch file to run a node responsible for publishing velocity information to the robot. see the code below:

   ```
   self.pubNode = rospy.Publisher('/cmd_vel', Twist, queue_size=1)
   self.msgTwistObj = Twist()
   self.msgTwistObj.linear.x = 0.2
   self.pubNode.publish(self.msgTwistObj)
   ```

   Here, we create a vairable called **pubNode** that is responsible for publishing Twist information to **cmd_vel** topic. And another variable called **msgTwistObj** that holds Twist message values. As you can see in the last command , we added 0.2 value to the x linear position of the robot.The last command, we used '**publish()**' function to publish the new updated message values.

7. Another topic we use is **/scan** topic, we use this topic to get the laser information, readings from the robot. For example, the distance between the robot and a wall in the environment. The message used is of type LaserScan. See the commands below:

   ```
   self.subNode = rospy.Subscriber('/scan', LaserScan, self.callback)
   self.laserMsg = LaserScan()
   ```

   In our code, we create variable called **subNode** to subsicribe to /scan topic and variable named **laserMsg** which holds a laser information and readings. The callback parameter in the Subscriber function is a function that has the updated laser information. see command below:

   ```
   def callback(self, msg):
   self.laserMsg = msg
   self.laserMsg.ranges
   ```

   So, whenever the robot moves, the variable laserMasg will be updated. One of the useful information we can obtain from laserMsg is the ranges parameter. The range parameter gives the distance between the robot and an object in the environment. The figure 2.3 visualizes and simulates the movement of robot.

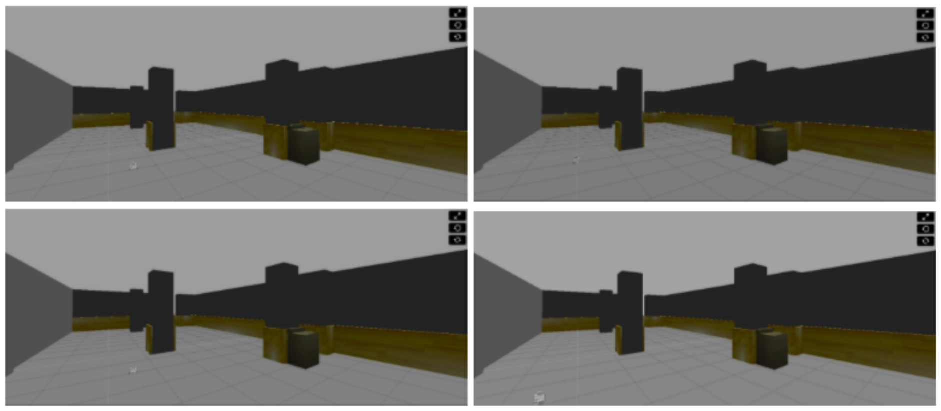


Figure 2.3: The movement of small robot in white colour can be seen. The screenshots are recorded on temporal scale. The first image captured is top left, scond one is top right, then bottom left and finally bottom right.

In this section, the fundamental cocepts of nodes and topics were learned to achieve the task. We also familiarized ourselves with the robot turtlebot 3, with the help of lecture and internet.

2.1.2 Task 2: Mapping & localization

To start the autonomous navigation process, the robot must have a map of the environment to be able to recognize objects, walls where it will allow the robot to plan trajectories through environment. So we first start with the mapping task. ROS Navigation[4] was helpful course.

2.1.2.1 Mapping

We mapped the environment by the following steps:

1. Launch TurtleBot3.

2. To start of the mapping process we need to use **gmapping** package that provides **slam_gmapping** node. This node is implementing the gmapping **SLAM** algorithm. It creates a 2D map of the environment using the data the Robot is providing during movement like laser data, in which it will be transformed to an **Occupancy Grid Map (OGM)** data format (**nav_msgs/OccupancyGrid.msg**) where it represents a 2-D grid map and each cell of the grid represents the occupancy (if the cell is completely occupied or completely free). Start the mapping process by adding this command to the launch file:

```
<node pkg="gmapping" type="slam_gmapping"
            name="turtlebot3_slam_gmapping" output="screen">
```

3. In the mapping process, an important tool is used called RViz. It will help us in visualizing the map around the robot, it will allow us to see what obstacles the robot is surrounded by in the environment. To launch Rviz. Execute this command:

```
rosrun rviz rviz
```

   It can be seen in the figure 2.4, In the left side, we can see the displays which can be added by us. we are interested in three displays which are:

   (a) **Map**: To visualize the map. Topic is /**map** where it has message of type Occupancy Grid Map OGM,

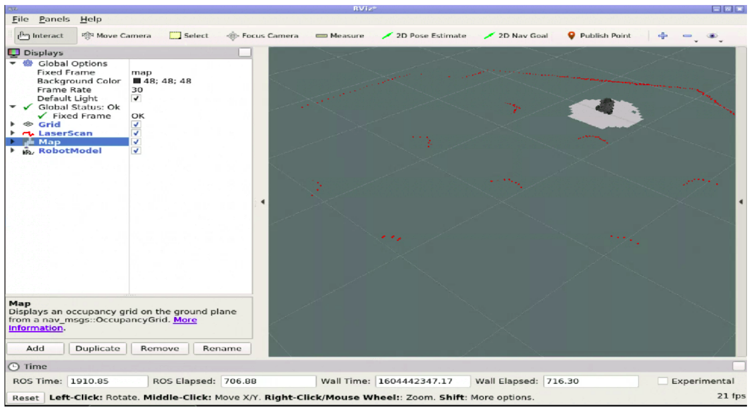
   ```
   nav_msgs/OccupancyGrid.msg
   ```

   (b) **LaserScreen**: To visualize what the Laser on the robot is detecting. Topic is /**scan.**

   (c) **RobotModel**: To localize the Robot on the map.

4. After launching **slam_gmapping** and **RViz**, we can start moving the robot by executing **Keyboard** control command:

```
roslaunch turtlebot3_teleop turtlebot3_teleop_key.launch
```



```
Global Options
  Fixed Frame          map
  Background Color     ■ 48; 48; 48
  Frame Rate           30
  Default Light        ✓
Global Status: Ok
  Fixed Frame          OK
Grid                   ✓
LaserScan              ✓
Map                    ✓
RobotModel             ✓

Map
Displays an occupancy grid on the ground plane
from a nav_msgs::OccupancyGrid. More
Information.

ROS Time: 1910.85   ROS Elapsed: 706.88   Wall Time: 1604442347.17   Wall Elapsed: 716.30   Experimental
Left-Click: Rotate. Middle-Click: Move X/Y. Right-Click/Mouse Wheel:: Zoom. Shift: More options.   21 fps
```

Figure 2.4: The figure demonstrates the simplest and basic map around the robot. The rviz command actully enables the lidar and sends laser beams all around the robot. These laser beams are reflected by the obstacle or wall and come back. Since the time of laser comming back and speed of light is known, the distance of obstacle around it can be easily calculated by the formula $distance = velocity \times time$. The map of obstacle around is represented by red lines/dots.

5. After moving the robot around all the places needed we should see the map fully occupied in **rviz** as shown in figure 2.5.

6. The map can be saved using **map_server** package, it includes **map_savor** node which will allow us to access the map data. Execute this command:

   ```
   rosrun map_server map_savor -f <file_name>
   ```

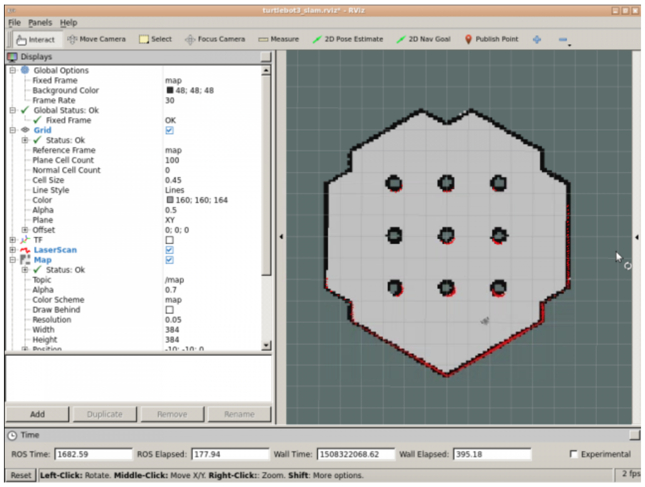
   After executing it, It will generate two files that are as follows:

   (a) **file_name.pgm**: PGM stands for **Portable Gray Map** where it contains the Occupancy Grid Map(OGM) data. If we download the file and open it, it will look like figure 2.6:

   (b) **file_name.yaml**: This file contains the meta data of the generated map which contains these parameters, image,resolution, origin, occupied_thresh, free_thresh,negate.

2.1.2.2 Localization

After creating the map, the next step is to locate the robot in the environment (created map). We can define localization as the process of finding the location of the robot in respect with the environment. For now, we have the map of the environment created, and we have sensors located on the robot which will observe the environment. We intend to estimate the coordinates and angles of whereabouts of the robot in the environment. The localization guide is explained in steps as follows:



```
Global Options
  Fixed Frame           map
  Background Color      ■ 48; 48; 48
  Frame Rate            30
✓ Global Status: Ok
  ✓ Fixed Frame         OK
Grid                    ✓
  ✓ Status: Ok
  Reference Frame       map
  Plane Cell Count      100
  Normal Cell Count     0
  Cell Size             0.45
  Line Style            Lines
  Color                 ▨ 160; 160; 164
  Alpha                 0.5
  Plane                 XY
  Offset                0; 0; 0
TF                      ☐
LaserScan               ✓
Map                     ✓
  ✓ Status: Ok
  Topic                 /map
  Alpha                 0.7
  Color Scheme          map
  Draw Behind           ☐
  Resolution            0.05
  Width                 384
  Height                384
  Position              -10; -10; 0
```

```
ROS Time: 1682.59   ROS Elapsed: 177.94   Wall Time: 1508322068.62   Wall Elapsed: 395.18   ☐ Experimental
Left-Click: Rotate. Middle-Click: Move X/Y. Right-Click:: Zoom. Shift: More options.
```

Figure 2.5: The figure shows the map of the environment around the robot. It is viewed from top of the robot. it can be said roof view.

1. To apply localization, we use AMCL (Adaptive Monte Carlo Localization) package which provides amcl node. It is a localization system that implements Kullback-Leibler(Monte Carlo) algorithm which uses an adaptive particle filters to track the position of the robot in respect with the environment. Here it is important to know about Monte Carlo Algorithm[1]. When the robot starts moving, this algorithm starts generating Particles and then uses the obtained sensor information (/**scan**) to create an optimized robot movement (make all the created Particles head to the right direction). The table in figure 2.7 describes the topics subscribed and published by AMCL. .

   (a) **map**: amcl subscribe to map topic to get the map data (OGM), to used it for localization.

   (b) **scan**: To have the updated scan readings.

   (c) **tf**: Transform topic which is necessary to provide the relationship between different reference frames. For example, translate from the base_laser (reference frame of the laser) coordinate frame to base_link(reference frame for the center of the robot) coordinate frame.

   (d) **amcl_pose**: amcl node publishes the position of the robot in the environment to the amcl_pose topic.

   (e) **particlecloud**: amcl publishes the particle cloud of arrows created by the system to measure the uncertainty of the robot current position as seen in the figure 2.8.

[1]Monte Carlo Algorithm uses particle filter to estimate the state (Localization) of the robot. The distribution of all likely states is represented and based on that, hypothesis is constructed which estimates where the robot is located (high probability state).

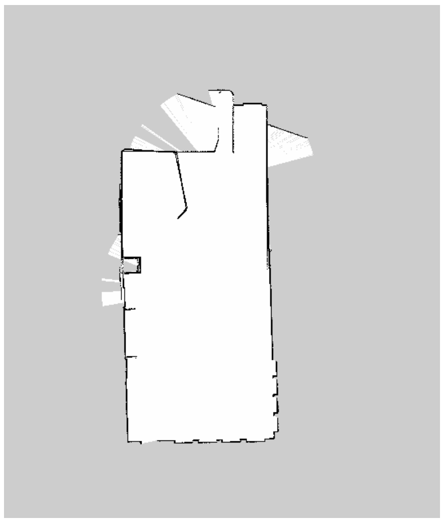
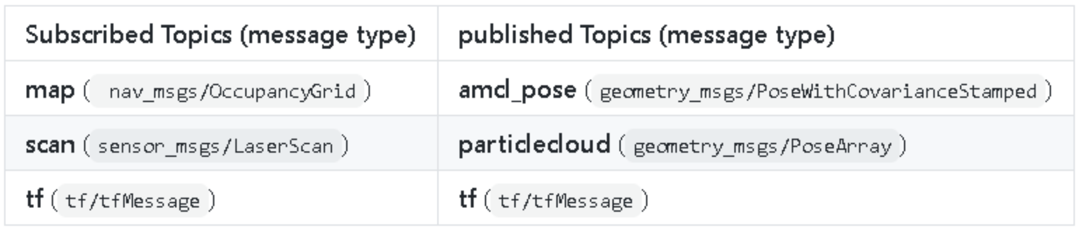


Figure 2.6: The map as shown from top. Each cell ranges from 0 to 100 integer value where 0 means completely free and not occupied, 100 is completely occupied



Subscribed Topics (message type)	published Topics (message type)
map (`nav_msgs/OccupancyGrid`)	**amcl_pose** (`geometry_msgs/PoseWithCovarianceStamped`)
scan (`sensor_msgs/LaserScan`)	**particlecloud** (`geometry_msgs/PoseArray`)
tf (`tf/tfMessage`)	**tf** (`tf/tfMessage`)

Figure 2.7: The

2. To launch amcl and start the localization process, we create a launch file which includes:
   (a) Launch TurtleBot3 applications:
   ```
   <include file="$(find turtlebot3_bringup)
           /launch/turtlebot3_remote.launch" />
   ```
   (b) Call our generated map file:
   ```
   <arg name="map_file" default
           ="$(find pkg_name)/maps/map.yaml"/>
   ```
   (c) Run map server node with our generated map:
   ```
   <node name="map_server" pkg="map_server" type
           ="map_server" args="$(arg map_file)" />
   ```

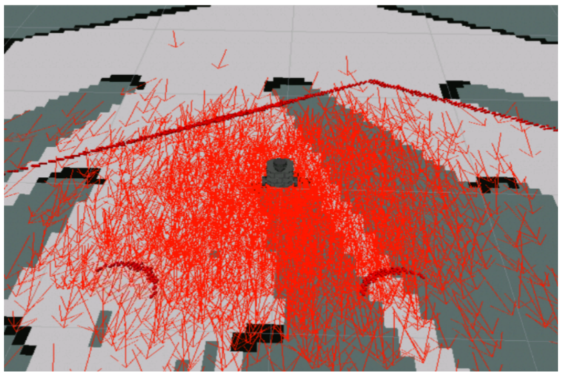


Figure 2.8: The red arrows displayed using Rviz,add **PoseArray** display which subscribe to **PointCloud** topic

(d) Launch amcl node:

```
<node pkg="amcl" type="amcl" name="amcl">
```

(e) AMCL node depends on some parameters like min_particles, max_particles. These parameters decide the number of particles used in the localization process. We add the parameters in the launch file:

```
<param name="min_particles" value="500"/>
<param name="max_particles" value="3000"/>
```

3. We use Rviz 2D Pose Estimate button to send the navigation system an initial pose of the robot. The last thing is to start moving the robot around the environment by using turtlebot3_teleop package.

2.1.2.3 Task 3: Path Planning

The path planning task is achieved in the following way.

1. After creating a map, localize the robot, we need to plan a path, trajectory for the robot to follow to reach a specific goal while avoiding obstacles along the way. To achieve this, we need to use move_base package which provides move_base node. This package is the base of the navigation task where it connects all the navigation components. See the figure 2.9.

2. The table in figure 2.10 describes topics, the message and their description. SimpleActionServer provides topics like goal(**move_base_simple/goal**), Feedback (**/move_base/feedback**) and Result(**/move_base/result**) etc. These topics are described as follows:

   (a) **FeedBack**: Keeps updating the server about the current information of the robot along the path (current position, laser information).For

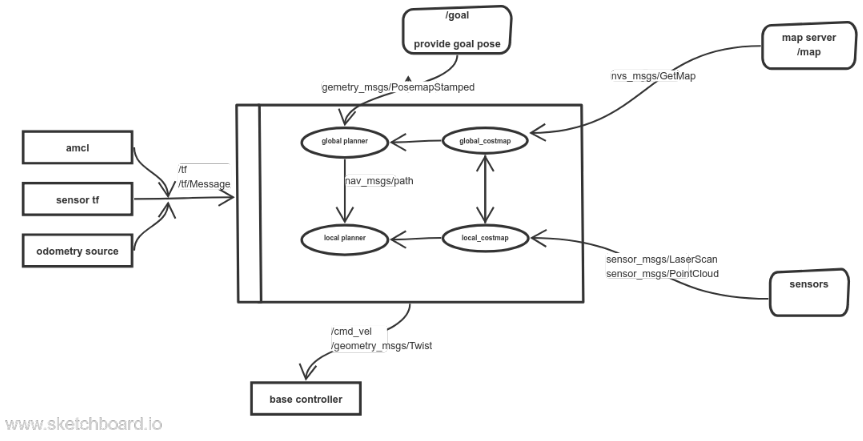


Figure 2.9: The figure shows how the **move_base** node interact with other system components. The node implements **SimpleActionServer** from **actionlib** package with message of type **gemetry_msgs/PoseStamped**. The Action server provides **/goal** topic that will provide the **move_base** node with goal position. . The figure was drawn on sketchboard

example, if we create a goal and the robot start to move toward the goal. If we display the message from the FeedBack topic we will see real time updated information in the terminal. Execute this command:

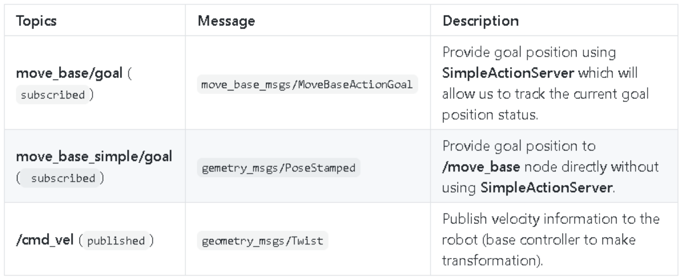
```
rostopic echo move_base/feedback
```

(b) **Result**: It is sent only once, a final pose of the robot is sent by the server to the **move_base** node When the robot reaches the goal.

3. We can see in the Navigation Task Figure above, there are parameters required to be loaded to the /move_base node. The parameters are described as follows:

   (a) **Global Planner**: The new goal, when received by move_base node, will be sent to the Global Planner. It will be responsible for building a safe plan for the robot to avoid collisions, obstacles along the way to the goal. It is called global because it starts planing the path from the beginning for the whole map and it doesn't depend on the current laser information provided by the robot.

   (b) **Local Planner**: The local planner uses only the information currently provided by the robot sensor readings and plans a path within a small area surrounding the robot position. When the next set of information come in, it will plan the new path.



Topics	Message	Description
move_base/goal (`subscribed`)	`move_base_msgs/MoveBaseActionGoal`	Provide goal position using **SimpleActionServer** which will allow us to track the current goal position status.
move_base_simple/goal (`subscribed`)	`gemetry_msgs/PoseStamped`	Provide goal position to **/move_base** node directly without using **SimpleActionServer**.
/cmd_vel (`published`)	`geometry_msgs/Twist`	Publish velocity information to the robot (base controller to make transformation).

Figure 2.10: The table illuminates some Topics and their description.

(c) **Costmap parameters (local & global)**: the costmap parameters are responsible for storing the information related to obstacles in the environment(map). The global costmap is a static map used to store information about the whole map to be used for global planning where local costmap is used to store local information (the small area surrounding the robot position) to be used for local planning. The figure 2.11 and 2.12 show these maps for visualization.

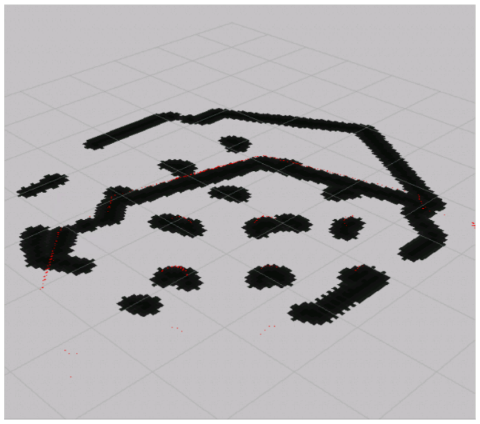


Figure 2.11: Global Costmap

4. To implement path planning with obstacle avoidance, we create a launch file where it includes the **map_server** package to get the map data, **amcl** package to localize the robot , and **move_base** package with its parameter configuration. As explained, move base node requires some parameters to be loaded. To configure and add move base node, we execute the following commands:

   (a) To launch the node

```
<node pkg="move_base" type="move_base"
```

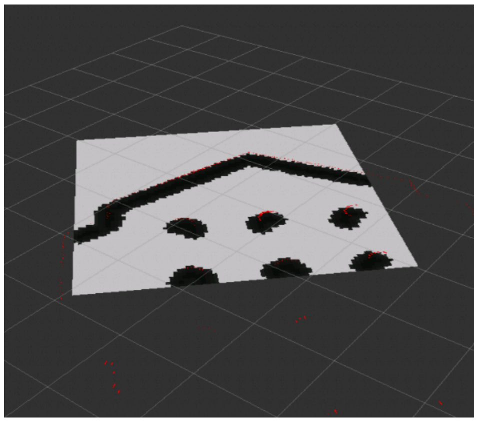


Figure 2.12: Local Costmap

```
            respawn="false" name="move_base" output="screen">"
```

(b) Load the required files (local & global costmaps | local & global planners). All the required parameters are included in these yaml files:

```
<rosparam file="$(find t3_navigation)/param/costmap_common_params_$
                (arg model).yaml" command="load" ns="global_costmap/>
<rosparam file="$(find t3_navigation)/param/costmap_common_params_$
                (arg model).yaml" command="load" ns="local_costmap"/>
 <rosparam file="$(find t3_navigation)
                /param/local_costmap_params.yaml" command="load" />
<rosparam file="$(find t3_navigation)
                /param/global_costmap_params.yaml" command="load" />
<rosparam file="$(find t3_navigation)
                /param/move_base_params.yaml" command="load" />
<rosparam file="$(find t3_navigation)
                /param/dwa_local_planner_params.yaml" command="load" />
‘‘‘
```

5. To create a goal, we can use Rviz.
   (a) Launch move_base node. The map should be created and the robot localized. (Turtlebot3)
   (b) Run Rviz and add add all necessary displays to visualize the navigation process.
   (c) To visualize the **local costmap**, **global costmap**, we add two Map display elements and attach them to **/move_base/local_costmap/costmap** and **/move_base/global_costmap/costmap** topics respectively.

(d) To visualize the local plan, global plan, we add two Path display elements and attach them to /**move_base**/**DWAPlannerROS**/**local_plan** and /**move_base**/**NavfnROS**/**plan** topics respectively.

(e) In Rviz, we choose **2D Pose Estimate** button to initialize the robot pose. Then we choose the **2D Nav Goal** button and click on the position where we want our turtlebot3 robot to move (goal). Figure 2.13 demonstrate navigation through visuals.

(f) After creating a goal, a goal message **(gemetry_msgs/PoseStamped)** will be published to /**move_base**/**goal** topic.

(g) The goal message we published to /**move_base**/**goal** topic will be received by **SimpleActionServer** which is implemented in the move-base node. So, the goal information will be received by the **move_base** node with goal topic provided by **SimpleActionServer** with message type **move_base_msgs/MoveBaseActionGoal.**

(h) We can run this command to see what has been published to goal topic:

```
rostopic echo /move-base/goal
```

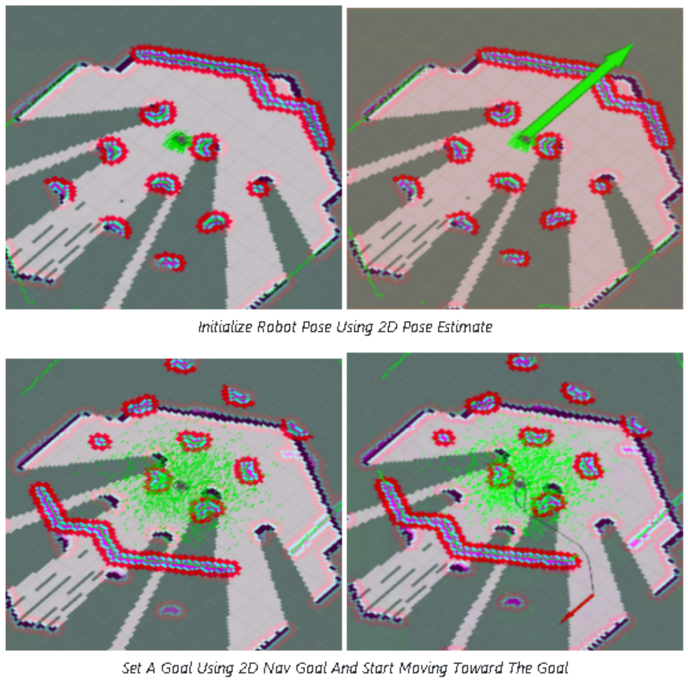

Initialize Robot Pose Using 2D Pose Estimate
Set A Goal Using 2D Nav Goal And Start Moving Toward The Goal

Figure 2.13: Visualization of simple navigation to a goal.

6. We can create a goal by directly publishing to the goal topic. By executing this command:

   ```
   rostopic pub /move_base/goal/ move_base_msgs/MoveBaseActionGoal
   ```

7. Another way we can create our goal is by creating an action client program(node) that send a goal to move_base.

   (a) Initialize the node and create Publisher to publish the goal to move_base node.

   ```
    rospy.init_node("GoalSender")
    pub = rospy.Publisher("move_base/goal", MoveBaseActionGoal)
   ```

   (b) Import all the move base package messages. Execute the following:

   ```
   from move_base_msgs.msg import
   ```

   (c) Create a function to send the goals. Inside the function we initialize a goal object from **MoveBaseActionGoal** and then we configure the goal parameters (position). Finally, we call the function to publish the goal. The code is as follows:

   ```
   def GoalSender(publisher):
           goal = MoveBaseActionGoal()
       goal.goal.target_pose.header.frame_id = "map"
       goal.goal.target_pose.pose.orientation.w = 1.0
       goal.goal.target_pose.pose.position.x = 0
       goal.goal.target_pose.pose.position.y = -5
       publisher.publish(goal)
   GoalSender(pub)
   ```

   (d) The **MoveBaseActionGoal** has a goal parameter of type **MoveBaseGoal.msg** which has the **target_pose** parameter of type **geometry_msgs/PoseStamped.msg.** that will allow us to create a goal.

8. To send goals using **SimpleActionServer**.

   (a) We create a node and import the necessary packages like actionlib.

   ```
   import actionlib
   from move_base_msgs.msg import MoveBaseAction, MoveBaseGoal
   ```

   (b) Initialize an action server. So we create a **move_base** action client (**SimpleActionClient**).

   ```
   client = actionlib.SimpleActionClient('move_base',MoveBaseAction)
   # this command to wait for the server to start listening for goals.
   client.wait_for_server()
   ```

(c) Create a goal to send to the server

```
# Create a goal with the MoveBaseGoal directly
goal = MoveBaseGoal()

# Configure the parameters
goal.target_pose.header.frame_id = "map"

goal.target_pose.header.stamp = rospy.Time.now()

# Translate 0.8 meters along the x axis w.r.t. map reference frame
goal.target_pose.pose.position.x = 0.8

# No rotation of the mobile base frame w.r.t. map reference frame
goal.target_pose.pose.orientation.w = 1.0
```

(d) Send the goal to the action server we created.

```
# Sends the goal to our action server.
client.send_goal(goal)
# Waits for the server till the end of the process.
wait = client.wait_for_result()
```

9. Now Turtlebot3 is able to navigate through the environment and follow a safe path without any obstacle collisions.

2.1.2.4 Task 4 : Create Way Points

If we want the robot to pass through multiple waypoints(goals) before reaching its destination, we can use a package called **follow_waypoints** where it will be responsible for storing these waypoints. This package will publish the stored waypoints to **move_base** node as a sequence and then the robot will reach the goal by navigating through all the waypoints. The following step by step guide provides implementation for achieving this task.

1. To install this package, we need to clone its GITHUB repository into our catkin src folder. We clone the repository by following command in terminal:

   ```
   git clone https://github.com/danielsnider/follow_waypoints.git
   ```

   After installing the package we need to be build our **catkin_ws** working space again, so we change the current directory to **catkin_ws** and then we execute: **catkin_make source /devel/setup.bash** Now, we have **follow_waypoints** package ready.

2. To start the waypoint server, we first start the navigation task process we implemented before:

```
roslaunch <our navigation package> <our launch file.launch>
```

3. Now we launch follow_waypoints:

   ```
   roslaunch follow_waypoints follow_waypoints.launch
   ```

4. The waypoint server listen to **initialpose** (**amcl** and **follow_waypoints** subscribe to this topic) that is used to initialize the robot with message type

   ```
   geometry_msgs/PoseWithCovarianceStamped
   ```

   The server will store all the waypoints and then will provide it to **move_base** node to start navigating through all the specified waypoints.

5. After launching all the necessary packages, we can start creating waypoints using **Rviz** and we add **PoseArray** display element(see Figure 2.14) and we add **waypoints** topic to this display. We run **Rviz** tool with already implemented configuration this time.

   ```
   rosrun rviz rviz -d `rospack
             find turtlebot3_navigation`/rviz/turtlebot3_nav.rviz
   ```

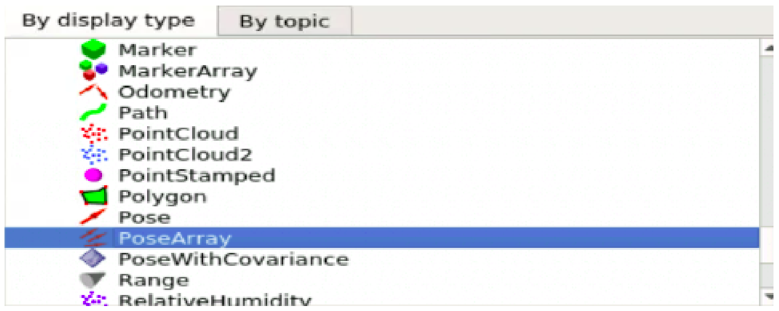


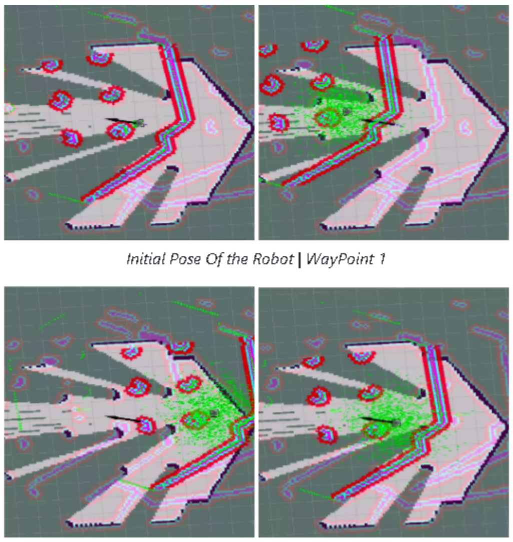
Figure 2.14: PoseArray display in Rviz

   (a) Use 2D pose Estimate to localize the Robot.
   (b) Use 2D Nav Goal to add goals (waypoints). See figure 2.15.

6. Now that we added our waypoints . We should start the **path_ready** topic **(follow_waypoints)** subscribes to this topic to initialize the process to follow our waypoint), then it will start sending the waypoints we created to **move_base** node. This topic has message of type **std_msgs/Empty**. To start the **path_ready** topic, we execute:

   ```
   rostopic pub /path_ready std_msgs/Empty -1
   ```

   After that, our Robot will start following the created waypoints.



Initial Pose Of the Robot **|** *WayPoint 1*

WayPoint 2 **|** *WayPoint 3(The last waypoint we choose should be in the same position as the initial Robot pose)*

Figure 2.15: This figure shows sequence of way points or goals where the robot is heading

7. If we want to create our own custom sequence of waypoints and implement the navigation through all the points autonomously, let's execute the following:

   (a) First, we set up waypoints locations. So, what we need is a coordinates of the waypoints with respect to map reference frame. To implement that, we create a list or a dictionary of our waypoint coordinates.

   ```
   locations = Dict()
   locations['waypoint1'] = Pose(Point(0.5, 4.0, 0.000),
           Quaternion(0.000, 0.000, 0.223, 0.975))
   locations['waypoint2'] = Pose(Point(−1.3, 2.382, 0.000),
           Quaternion(0.000, 0.000, −0.670, 0.743))
   locations['waypoint13'] = Pose(Point(−3.756, 6.401, 0.100),
           Quaternion(0.000, 0.000, 0.733, 0.680))
   ```

   We can get these coordinates by using Rviz tool. Press the 2D Nav Goal button and click on the location of the waypoint we want. The coordinates will be displayed on the terminal.

   (b) Subscribe to move_base action server.

   ```
   import actionlib, rospy
   from move_base_msgs.msg import MoveBaseAction, MoveBaseGoal
   from geometry_msgs.msg import PoseWithCovarianceStamped,
   ```

```
PoseArray,Pose,Quaternion,,Twist
# subscribe to action server
client = actionlib.SimpleActionClient('move_base',MoveBaseAction)
# this command to wait for the server to start listening for goals.
client.wait_for_server()
```

(c) Create a variable to hold the initial position of the robot w.r.t the map.

```
 initial_pose = PoseWithCovarianceStamped()
# This line of code is used when to get the initial position
# using RViz (The user needs to click on the map)
rospy.wait_for_message('initialpose', PoseWithCovarianceStamped)
self.last_location = Pose()
```

(d) Subscribe to **initialpose** with callback function called **update_initialpose_callback** to update the initial_pose variable with the current initial pose.

```
# Subscribe to initialpose topic
rospy.Subscriber('initialpose', PoseWithCovarianceStamped,
                 update_initialpose_callback)
# callback function to update the initial_pose vaule
                 to the current initial pose.
def update_initialpose_callback(self, initial_pose):
initial_pose = initial_pose
```

(e) Start the process of sending goals(waypoints) to move_base action server. We take a sequence of waypoint locations from the dictionary we created and then we iterate over the sequence to navigate through all waypoints.

```
# Set up a goal location
goal = MoveBaseGoal()
#location = ['waypoint1', 'waypoint2', 'waypoint3']
goal.target_pose.pose = locations[location]
goal.target_pose.header.frame_id = 'map'

#For the user, to know where the robot is moving (to which waypoint)
rospy.loginfo("Going to: " + str(location))

# Send the goal(current waypoint) to move_base action server
client.send_goal(goal)
```

Note that the custom waypoints program is not a fully complete code, or a correct program. but a sample of how we could implement the process. We are confident that our approach will work. It might not be the most efficient or optimized solution though.We are open to new ideas. **Sky is the limit**!.

Chapter 3

Conclusion and Recommendations

This technical survey provides a brief solution in the form of simulation for the navigation of Robot in an indoor environment. The simulation was run on Gazebo in 'The Construct'. The project is divided into four tasks which are Robot Control, Mapping and Localization, Path Planning (Obstacle Avoidance) and Way Point Navigation. The three most important and useful packages are gmapping, amcl and move_base. We tested Navigation tutorials on multiple Maps provided by 'The Construct' in multiple courses. The Construct courses were really helpful and guiding in understanding the problem and implementing the solution. With the evidence of working simulations and visualizations, we are confident that if we apply our algorithm and tutorials on the real robot, we will be successful in navigating the robot and hence integrating different ROS concepts and packages on Robots. We intend to apply our knowledge and experience in ROS for the navigation of Robot on simulated map of Costa Coffee at Barcelona provided in the project description document. However, we solemnly hereby do not claim that our solution is efficient, optimized or validated by practical experimentation or expert. We are open to any new ideas and we intend to learn, relish making mistakes and gain expertise in Robotics.

We are convinced that ROS is a very powerful tool in robotics however it is not easy to gain expertise. It is a universe of information in itself. We felt a lot of pressure for our tight schedules and the constraint of timing was felt gravely. We tried our best to practice and learn and what we have is a birds eye view of ROS. We recommend that the course of ROS be taught in two semesters keeping the intensity of work less. The Construct is a wonderful platform and it makes learning of ROS quite smooth and efficient. However, it has some fundamental issues that make it less compatible. The Construct has stability issues. It is slow and sometimes it crashes. It is unstable, not trust-able regarding the privilege of saving progress. The Construct consumes a lot of Data and it is very difficult to use it on limited internet packages. We recommend them that there is a lot of room for improvement in computation and stability of website.

Due to COVID-19, we are working from home and do not have any access to real robots and we understand that. But, It is certain that we might not be able to gain expertise in working with real robots and we surely miss that opportunity.

Bibliography

[1] ALBERTO EXQUERRO. The construct, 2019.

[2] Patrick Goebel. *ROS By Example*. Lulu, 2013.

[3] Ricardo Téllez, Alberto Ezquerro, and Miguel Ángel Rodríguez. *ROS in 5 Days: Entirely Practical Robot Operating System Training*. Independently published, 2016.

[4] Ricardo Tellez, Alberto Ezquerro, and Miguel Angel Rodriguez. *ROS NAVIGATION IN 5 DAYS: Entirely Practical Robot Operating System Training*. Independently published, 2017.