MOBILE ROBOT NAVIGATION USING ROS FRAMEWORK

Project Laboratory Documentation

Project Members: Pranto Abir Shahriar (CSWK1K)

Yousha Mohammed Shamin Yeaser (WBDDO4)

Das Anjan Kumar (D42DQA)

Project Supervisor: Kiss Domokos

Weekly Work Diary

Week – 1: Learning introduction to ROS and ROS beginner tutorial.
Week – 2: Learning Gazebo beginner and intermediate tutorial.
Week – 3: Learning about Rviz and Transformation "Tf".
Week – 4: Learning about URDF models and creating one.
Week – 5: Learning how to connect Gazebo with ROS.
Week – 6: Learning Gopigo3 simulation and navigation in ROS with ROS navigation stack tutorials.
Week – 7: Learning about ROS actions.
Week – 8: Implementing Wall-follow algorithm as a ROS node.
Week – 9: Implementing Wall-follow algorithm as a ROS node.
Week – 10: Building a maze and creating a launch file for maze.
Implement an obstacle avoidance algorithm (VFF) as a ROS node
Implement RRT global planner as move_base plugin
Week – 11: Implementing wall follow algorithm as a maze solving algorithm.
Implement VFF algorithm as a move_base local planner plugin
Implement RRT global planner as move_base plugin
Week – 12: Creating and preparing the documentation and presentation.
Doing the presentation and submission of documentaion

- ## Installation

As all of the project members used windows operating system, so at first it was necessary to install the ubuntu linux operating system in the devices, all of us used the dual boot method for installation. And then we got acquainted with the operating system and learnt basic things.

- ## ROS Tutorials

We had to complete all the beginner level tutorial which helped us to know the basic operation of ROS as well as some structure of the system. We have used the ROS Noetic version. Some of the key takeaway points from these tutorials are as follows:

1. First we installed and configured the ROS environment in our local device which also helped us to learn how to operate the ROS system with the command window.
2. We learnt how to use source command and setup bash file, besides the structure of the bash file
3. Then we learned to create ROS workspace and the use of mkdir, cd and catkin_make commands .The catkin_make command is a convenience tool for working with catkin workspaces. Running it the first time in the workspace, it creates a CMakeLists.txt link in your 'src' folder. Additionally, if we saw that a 'build' and 'devel' folder was created.
4. We also learned the use of roscd command which part of the rosbash suite. It allows to change directory (cd) directly to a package or a stack.
5. We made packages by using the catkin_create_pkg command , beside we learnt about package dependencies and how to write it with the command
6. We created nodes within the package in order to execute the file. We learnt that ROS nodes use a ROS client library to communicate with other nodes. Nodes can publish or subscribe to a Topic. Nodes can also provide or use a Service.
7. We got to know about the use of roscore and rosrun command
8. Used rostopic which allows to get information about ROS topics and rostopic echo which shows the data published on a topic
9. Also, we learnt about the structure of simple publisher and subscriber , simple service and client.

- **Gazebo Beginner and Intermediate Tutorials**

    - In the beginner gazebo tutorials we got a overview of the software and got to know that it is an open-source 3D robotics modelling and simulator, which is widely used in the robotics world
    - We also got acquainted with the graphical user interface of the software, the scene , panels, toolbars, the menu and also got to know about the mouse control
    - Then we learnt about the model editor , we used the toolbar, palette, insert tab and model tab and as a sample work we constructed a dummy vehicle
    - In the intermediate tutorials , we got into the details of SDF modeling using an example of construction of a rotating lidar sensor.
    - We learnt about SDF Model, adding inertia, joint, sensor
    - We tried some model appearance, like mesh acquisition, creating model structure, using the meshes, use of textures
    - We also added noise to the sensor, and followed by uploading and control plugins

- **RViz Introduction**

RViz is a 3D Visualization Environment specifically for robot development.In gazebo we can create model and see what they are doing as the end result, but RViz lets us view what the robot is seeing, thinking and doing, which is very important for debugging. Because before applying the algorithm to a real robot it is necessary to simulate it using different maps and worlds.

There are two main ways to put data in RViz's world:

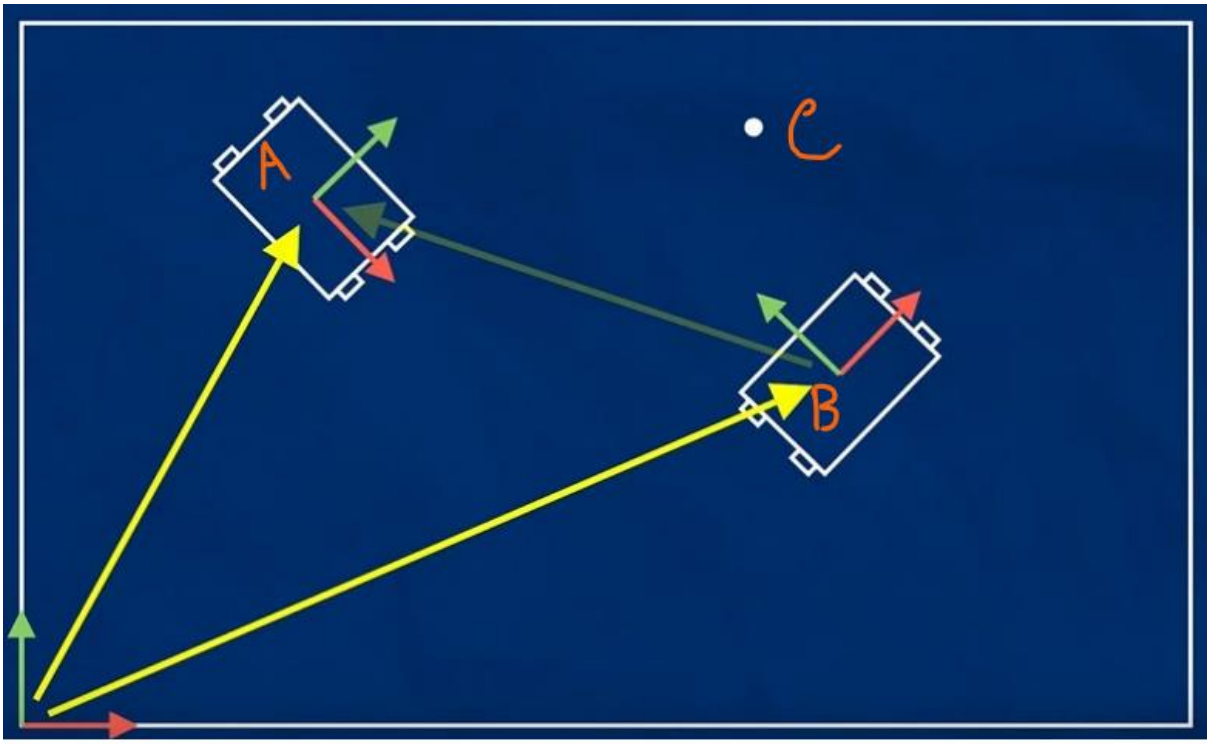
- RViz understands sensor and state information, like laser scan , cameras, point cloud , coordinate frames
- There is visualization markers that allows programmatic addition of various primitive shapes to the 3D view

  Also we got acquainted with the user interface if the software, learnt about displays and how to add it, display properties, status , built-in display types, configurations, cameras, views , co-ordinate frames , target frames, 2D nav goals etc.

- **Introduction to TF**

TF is used to transform coordinate frames. Whenever we think about a robot, we think about joints and links which makes the robot function properly. Each and every links and joints of the robot defines a specific part of the robot. These moving parts have their own local frame that need to be related to a global frame. Tf uses a branch style architecture to relate the different parts as they relate to the global frame.

TFs can be easily defined as a tree of FRAMES connected by TRANSFORMS. TFs have been around for over hundreds of years, but in recent years we have been using them in integration to ROS system. TF can be used to solve various sort of problems regarding localization and robot movements and positioning. A very common example can be, two robots in the same environment looking for an object.

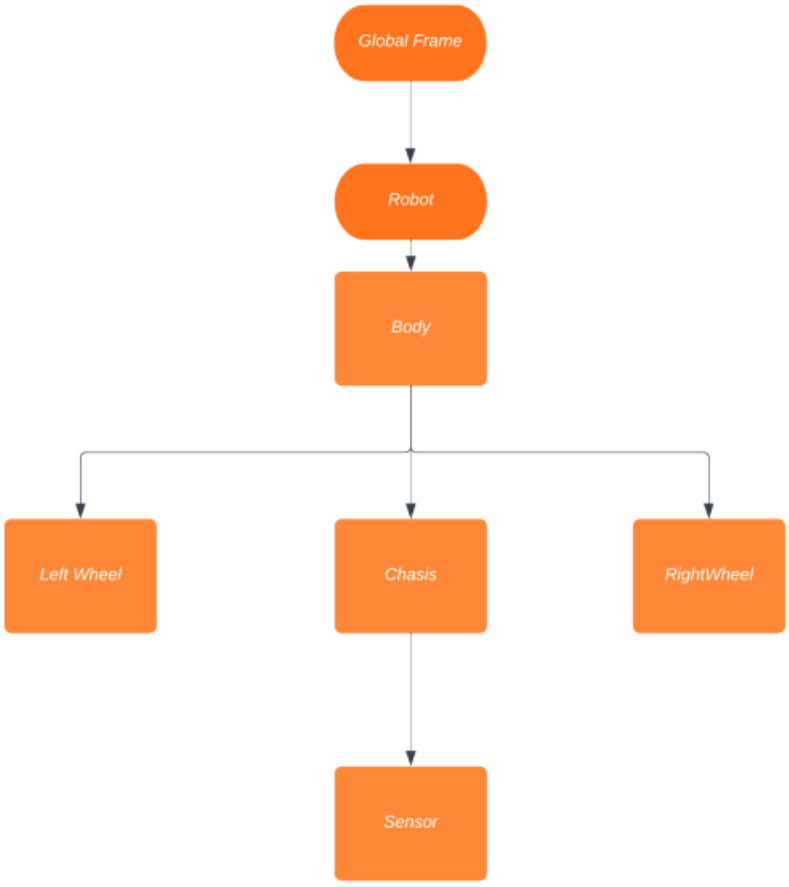


In this picture we can see A and B robots are looking for C object. Now the problem is we know how far is the object from B, we need to find how far is it from A. We have origin frame and two robot frame.

Now the main task of TF is to create transforms between the frames. In a nutshell, how can we change from one frame to another one. As long as there

is continuous line of transformation and connection we can solve this problem easily. Due to reduce the confusion to the connection of the TF problem, we always make a tree of TF respect to one of the origin frame. Meaning, we only create a tree where all the object will only have one arrow pointing towards it. It can have multiple going out, that will not be a problem. Now, if we know the frame of the B, we can transform it to origin frame and from the origin frame we can transform it to the A frame. And after that finding the location of the object from A will be easy.

To get a even more simple idea, if we imagine a robot with a BODY. The BODY is connected to LEFT and RIGHT wheel and has a chassis. And the sensor is on the top of the chassis. If we are to make such a TF tree of a robot is should look similar to this.



Regardless, TF is a very broad and connected topic. Operating an algorithm or working with robots in ROS system without TF is almost impossible. Thus the operational methodology and importance of TF surpasses its past reputation when it comes to controlling robots in virtual and real world environment.

- **URDF Files and Format**

URDF stands for Unified Robot Description Format. It's a Collection of description to let the system or ROS know how a robot looks like in real life. This concept is mostly useful to the researchers or engineers to play around with the simulation using gazebo or other software before actually acquiring the robot itself in real life. Another important perspective of having a URDF model is to simulate the algorithms without damaging the real robot.

URDF describes a robot as a tree of links, that are connected by joints. The links represent the physical components of the robot, and the joints represent how one link moves relative to another link, effectively defining the location of the links in space.

```
< ? xml version = "1.0" ? >
  < robot name = "origins" >
        < link name = "base_link" >
              < visual >
                    < geometry >
                    < cylinder length = "0.6" radius = "0.2" / >
                    < / geometry >
              < / visual >
        < / link >

        < link name = "right_leg" >
            < visual >
                < geometry >
                < box size = "0.6 0.1 0.2" / >
                < / geometry >
                < origin rpy = "0 1.57075 0" xyz = "0 0 -0.3" / >
            < / visual >
        < / link >


        < joint name = "base_to_right_leg" type = "fixed" >
              < parent link = "base_link" / >
              < child link = "right_leg" / >
              < origin xyz = "0 -0.22 0.25" / >
        < / joint >

  < / robot >
```

Here is a typical URDF file. In the following mostly used URDF tags will be explained:

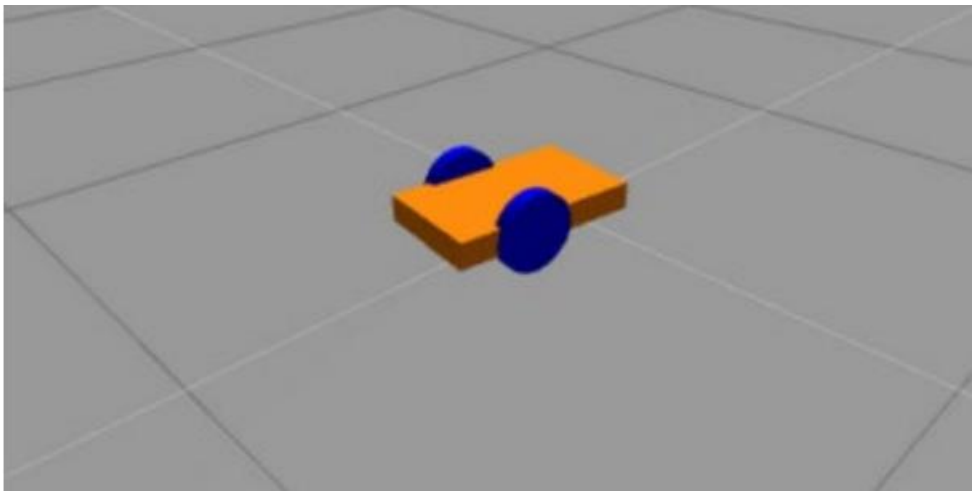
1) Robot Tag: We can see that there is robot tag, which is the main tag of a URDF file. Inside the robot tag, we may have link tags, joint tags, gazebo tags etc

2) Link Tag: which is to define the links of a robot with its body. Besides that it has some additional characteristics - the visual [This is what we see in RViz and Gazebo], collision [used for physics collision calculations], and inertial [determines how the link responds to forces] properties as well. These characteristics also have some sub characteristics.

3) Joint Tag: Even though we may imagine a robot consisting only of links, but the links themselves have their own location coordinate and definition of how they move relative to each other. To define those properties, we may use joint tags.

4) There are also some additional tags such as, material : To define the material color and so on, gazebo: properties to be used in gazebo simulator etc.

If we are done with all the description and coding, a simple robot with only two wheels (blue) and a chassis (orange) can be shown below



This robot has only two links and joints. The wheels have their own axis of rotation defined and connected to the body of the robot. Simulating in an virtual environment without a virtual description of the robot is almost impossible. Thus, URDF has significant importance in that perspective.

- **ROS Navigation Stack**

Navigation is an essential part of robotics, particularly in robot control. In order to send a robot to its correct goal point we have to navigate the robot. ROS navigation package make it simple. ROS navigation package provides us with a 2D

navigation stack that takes in information from odometry, sensor streams, and a goal pose and outputs safe velocity commands that are sent to a mobile base. It is simple on a conceptual level. But use of this in an arbitrary robot is quite complicated. It requires some additional setup for its use on robot. The requirements are:

- Robot operating by ROS.
- The robot has a transformation tree.
- The robot publishes sensor data in correct ROS message type.
- We have covered the transformation "tf" part in our previous discussions. We will continue from sensor information.

Sensor Information

.

The ROS navigation stack uses sensor information to avoid obstacle in its environment. It takes the information published in sensors/Laserscan or sensor_msg/PointCloud messages in ROS. Also, there are several sensors where there is ROS driver which takes care of this publishing messages.

Odometry Information

.

The navigation stack expects the odometry information to be published in the nav_msg/Odometry messages using transformation tf.

Base Controller

.

The navigation stack sends the velocity commands using a geometry_msgs/Twist message located in the base coordinate frame of the robot on the "cmd_vel" topic. So, a ROS node is used which will subscribe to the cmd_vel topic and take the xy,yz,theta values and convert it into motor commands to send it to mobile base.

Navigation Stack Setup

.

For using the navigation stack, we must set it up. We can set the navigation stack in the following steps:

- Creating a ROS package: We will create a package which will store all the configuration and launch files.
- Creating a robot configuration launch file: We must create a roslaunch file which will bring up all the hardware and transformation publishes that the robot requires for the stack setup.

- Cost map configuration: The navigation stack uses two cost maps to store the data about obstacle in the world. The cost maps should be pointed at the sensor topic in order to listen and update about obstacle. For this we will create a yaml file to store the common parameters.
- Global Cost map Configuration: Global cost map is responsible for global planning which indicates that it will create long term plans over the entire environment. We have to create a file which will contain the specific options for the global cost map. The global frame, robot base frame, update frequency and static map.
- Local Cost map: Local cost map is responsible for local planning which indicates that it will create short term plan over a specific environment. Same as the global cost map we will also create a yaml file to store the local options for the local cost map.
- Base local planner configuration: The base_local_planner is responsible for computing velocity commands to send to the mobile base of the robot given a high-level plan. We will need to create a yaml file to store the specific parameters for our specific robot.
- Finally, we need to create a launch file to bring all the configurations together including local and global cost map, base local planner, move base etc.

Move Base

- The move_base package provides an implementation of an action that, given a goal in the world, will attempt to reach it with a mobile base. The move_base is major component of the navigation stack. The move_base node links together a global and local planner to accomplish its global navigation task. The structure of move base is demonstrated below.

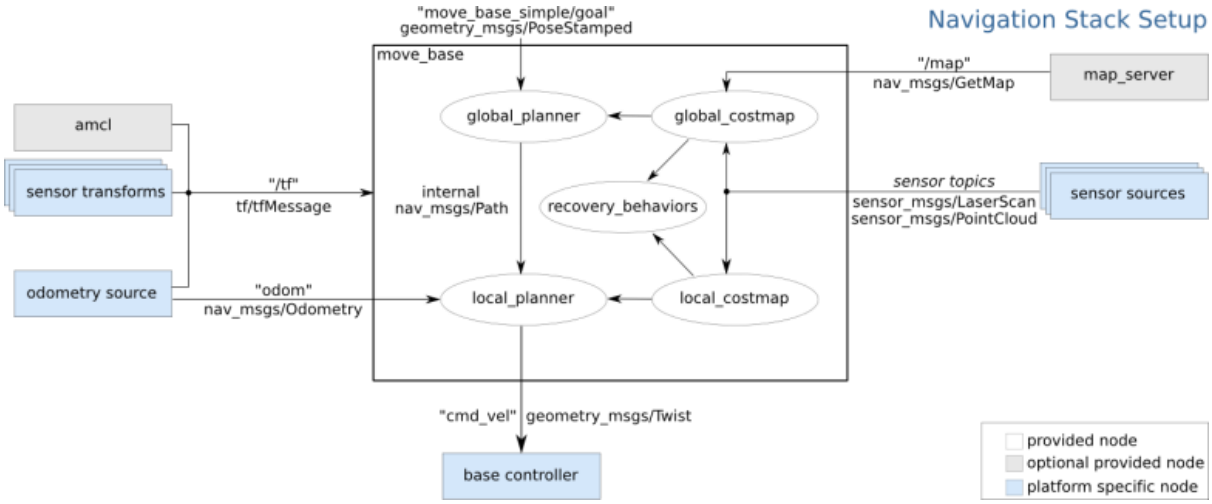


Figure 1: Move_base pacakge structure

The box represents the move_base of the navigation stack. The move_base node provides a ROS interface for configuring, running, and interacting with the navigation stack on a robot. The blue varies based on the robot platform, the gray is optional but are provided for all systems, and the white nodes are required but also provided for all systems.

•

Move_base Default Recovery Behavior

•

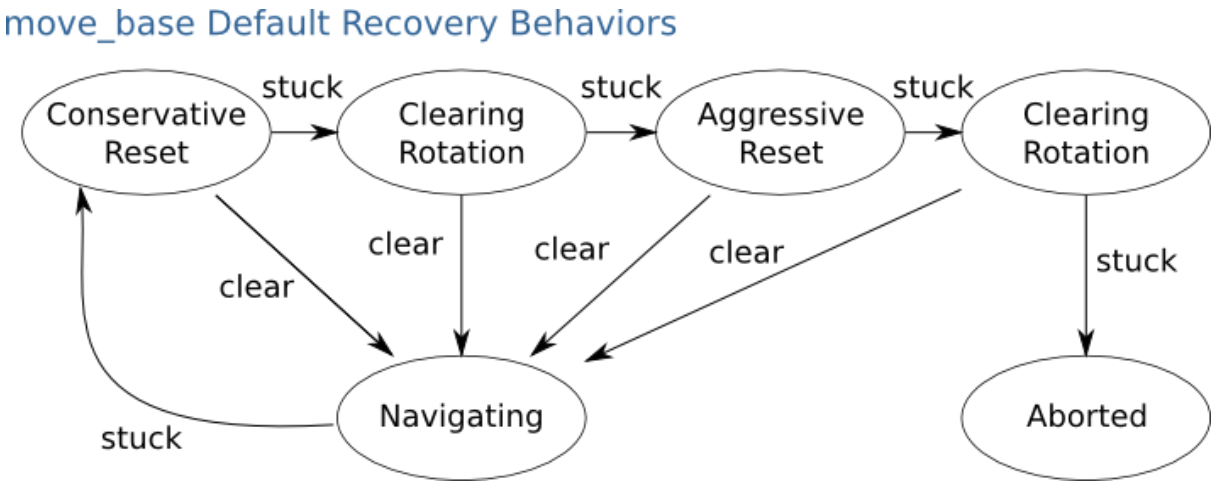


Figure 2: move_base default recovery behaviours

Running the move_base node on a robot that is properly configured (please see navigation stack documentation for more details) results in a robot that will attempt to achieve a goal pose with its base to within a user-specified tolerance. The move_base node may optionally perform recovery behaviors when the robot perceives itself as stuck. By default, the move_base node will take the following actions to attempt to clear out space:

First, obstacles outside of a user-specified region will be cleared from the robot's map. Next, if possible, the robot will perform an in-place rotation to clear out space. If this too fails, the robot will more aggressively clear its map, removing all obstacles outside of the rectangular region in which it can rotate in place. This will be followed by another in-place rotation. If all this fails, the robot will consider its goal infeasible and notify the user that it has aborted. These recovery behaviors can be configured using the recovery behaviors parameter and disabled using the recovery behavior enabled parameter.

Actions API of Move_base

The move_base node provides an implementation of the SimpleActionServer, that takes in goals containing geometry_msgs/PoseStamped messages. Action subscribed topics are as follows:

- move_base/goal (move_base_msgs/MoveBaseActionGoal): A goal for move_base to pursue in the world.
- move_base/cancel (actionlib_msgs/GoalID): A request to cancel a specific goal.
- move_base/feedback (move_base_msgs/MoveBaseActionFeedback): Feedback contains the current position of the base in the world.
- move_base/status (actionlib_msgs/GoalStatusArray): Provides status information on the goals that are sent to the move_base action.
- move_base/result (move_base_msgs/MoveBaseActionResult): Result is empty for the move_base action.

Published Topic is:

- cmd_vel (geometry_msgs/Twist): A stream of velocity commands meant for execution by a mobile base.

Move_base provides some services also:

- make_plan (nav_msgs/GetPlan): Allows an external user to ask for a plan to a given pose from move_base without causing move_base to execute that plan.
- clear_unknown_space (std_srvs/Empty): Allows an external user to tell move_base to clear unknown space in the area directly around the robot. This is useful when move_base has its costmaps stopped for a long period of time and then started again in a new location in the environment. - Available in versions from 1.1.0-groovy
- clear_costmaps (std_srvs/Empty): Allows an external user to tell move_base to clear obstacles in the costmaps used by move_base. This could cause a robot to hit things and should be used with caution. - New in 1.3.1

Global Path Planner as a plugin

- We can add a global path planner as a plugin in our robot using ROS. We can add a global planner as a plugin by following steps:
  i. The first step is to write a new class for the path planner that adheres to the nav_core::BaseGlobalPlanner.
  ii. Register the planner as BaseGlobalPlanner plugin:this is done through the instruction PLUGINLIB_EXPORT_CLASS(global_planner::GlobalPlanner, nav_core::BaseGlobalPlanner). For this it is necessary to include the library #include <pluginlib/class_list_macros.h>
  iii. Then implement the makePlan() method: The start and goal parameters are used to get initial location and target location, respectively. Then, 20 new dummy locations will be statically inserted in the plan in the for loop, then,

the goal location is inserted in the plan as last location. This planned path will then be sent to the move_base global planner module which will publish it through the ROS topic nav_msgs/Path, which will then be received by the local planner module.

Dynamic Window Approach (DWA) Local Planner

- The dwa_local_planner package provides a controller that drives a mobile base in the plane. This controller serves to connect the path planner to the robot. Using a map, the planner creates a kinematic trajectory for the robot to get from a start to a goal location. Along the way, the planner creates, at least locally around the robot, a value function, represented as a grid map. This value function encodes the costs of traversing through the grid cells. The controller's job is to use this value function to determine dx,dy,dtheta velocities to send to the robot.

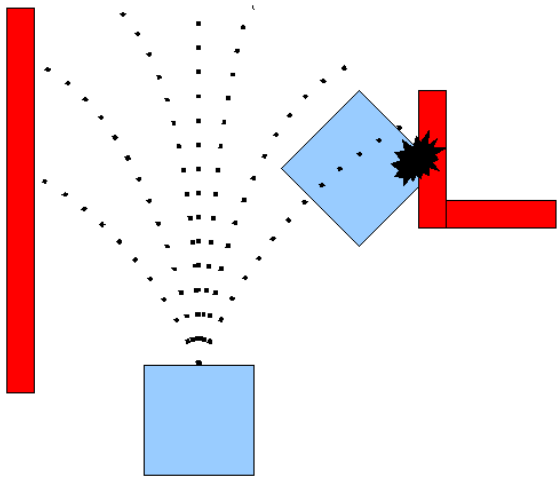


Figure 3: DWA Local Planner

The algorithm is as follows:

1. Discretely sample in the robot's control space (dx,dy,dtheta)
2. For each sampled velocity, perform forward simulation from the robot's current state to predict what would happen if the sampled velocity were applied for some (short) period of time.
3. Evaluate (score) each trajectory resulting from the forward simulation, using a metric that incorporates characteristics such as: proximity to obstacles, proximity to the goal, proximity to the global path, and speed. Discard illegal trajectories (those that collide with obstacles).
4. Pick the highest-scoring trajectory and send the associated velocity to the mobile base.

5. Rinse and repeat.

- **ROS Action library**

The actionlib stack provides a standardized interface for interfacing with preemptable tasks. Examples of this include moving the base to a target location, performing a laser scan, and returning the resulting point cloud, detecting the handle of a door, etc... The actionlib package provides tools to create servers that execute long-running goals that can be preempted. It also provides a client interface in order to send requests to the server.

- Client Server Interaction

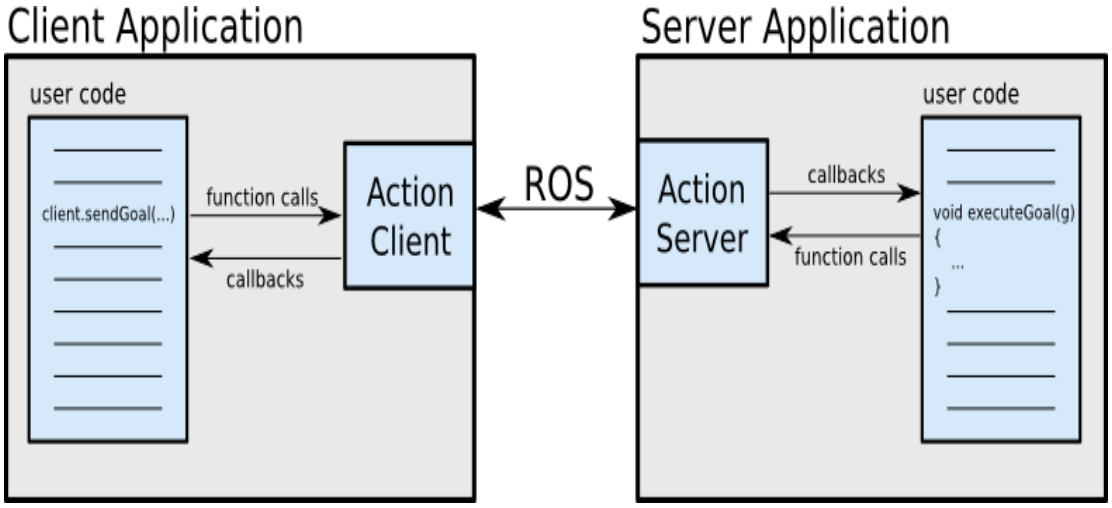


Figure 4: Client Server Interaction in ROS action library

The *ActionClient* and *ActionServer* communicate via a *"ROS Action Protocol"*, which is built on top of ROS messages. The client and server then provide a simple API for users to request goals (on the client side) or to execute goals (on the server side) via function calls and callbacks.

ROS action Specification

In order for the client and server to communicate, we need to define a few messages on which they communicate. This is with an *action specification*. This defines the Goal, Feedback, and Result messages with which clients and servers communicate:

- Goal: The goal is the target task that must be done by the server and which is requested by the client.
- Feedback: Feedback provides server implementers a way to tell an ActionClient about the incremental progress of a goal.
- Result: A result is sent from the ActionServer to the ActionClient upon completion of the goal.

- **Implementation of Wall Following and Maze Solving Algorithm**

Introduction

Robots are an essential part of modern technology. In an era of massive technological development, the robots occupy a lot of space. In the core of robot management there is robot navigation control. In this project we are implementing robot navigation tactics with the help of Robot Operating System (ROS). Robot Operating System is an open-source robotics middleware suite. Although it is not an operating system but it is a set of software framework for robot software development. In this part of the project work, I am going to implement wall following algorithm and maze solving algorithm to navigate my robot. The robot model that is used in this project is Gopigo3, a small three wheeled robot containing a rotating wheel and two wheels. The navigation is mostly done by the LIDAR sensor readings obtained by the LIDAR sensor placed above the robot. LIDAR shoots an array of laser rays all around it in 360 degrees with increment of 1 degree.

Task Description

In this part of this project my task is to implement wall following algorithm and maze solving algorithm into the robot using ROS in form of ROS node. I must implement an algorithm where the robot finds a wall and starts to follow it until it faces any obstacle. I can choose the left, right or both walls to follow. Then I must implement this wall follow algorithm in the robot such a way that it can solve a simple connected maze. The algorithms must be implemented as ROS nodes using C++ language.

Algorithms

The algorithm that I have used is the wall follow algorithm. The wall-follow algorithm states that if one keeps its hand on the wall and traverse the maze it will finally reach the exit [1]. Wall follow algorithm is best-known algorithm to solve simple connected maze. However, the wall follow algorithm may not achieve the required goal if the maze is not connected.

To implement the algorithm, I have followed the following blueprint:

- It starts with moving forward and searching for a wall at its right or left side.
- After getting the wall it starts to follow it until it faces any obstacle in front of it.
- Upon facing an obstacle, it checks the laser reading from the left and right side.
- At the same time, it also compares them the picks the larger one and turns toward that side.
- Then it starts to follow the wall on either left or right side depending on the turn it took.

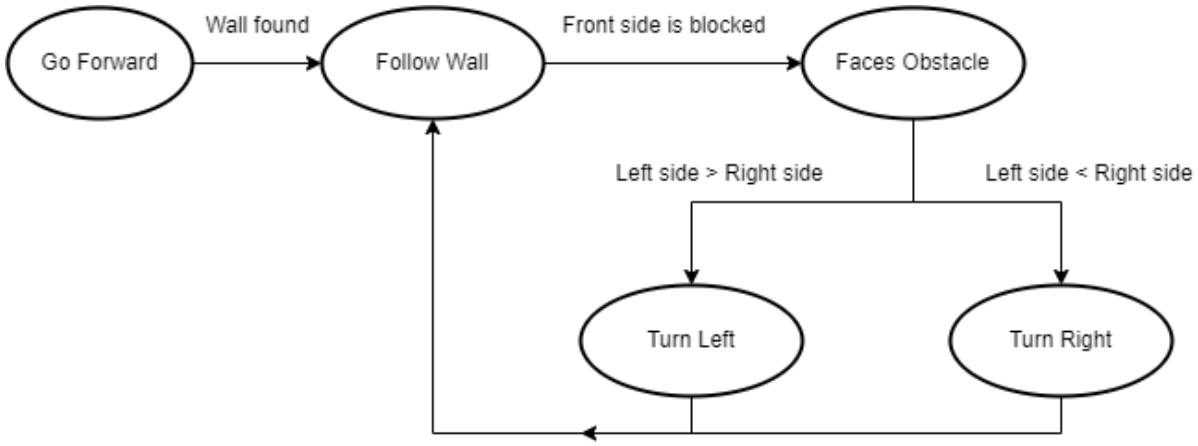


Figure 5: Flow Diagram for Wall Follow

Programming Implementation

To accomplish this task, I have taken 5 lasers reading from the laser reading array. I have chosen right side, left side, front side, top right side and top left side by specifying the index number from the laser range array. As the laser is positioned facing the right side with respect to the robot's orientation, So I have taken index 0, 89, 179, 104 and 44 as Right side, Front side, Left side, Top left side and Top right side respectively. I have set 0.5 m as the limit for checking an obstacle which means if the laser reading is 0.5 or less then it will detect it as an obstacle. I have implemented a function called laser callback to get the laser reading in a working form to analyze and compare them. The wall-follow node acts as a subscriber for the "scan" topic which provides me with the laser sensor data. This is the function where I have mostly implemented the algorithm. With the obtained laser data, I have compared them with each other to make the algorithm work.

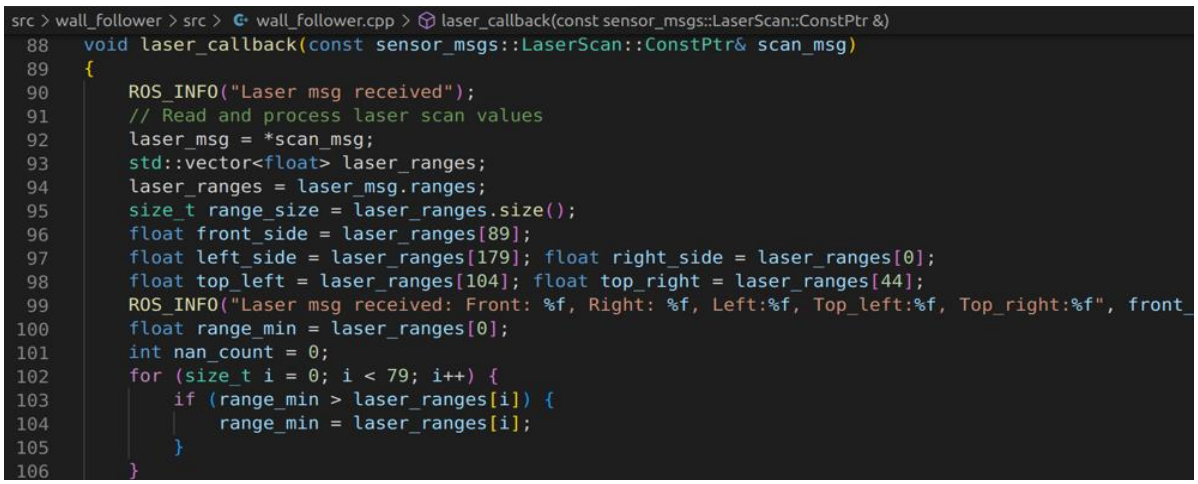


Figure 6: Assigning Laser data to variables

I have made the ROS node to communicate with the topic of my Gopiogo robot and publish the commands to navigate the robot. In the ROS node I have used the cmd_vel topic to publish my robot navigation commands such like: turn left, go forward etc. Cmd_vel topic provides us with a way to publish the coordinate for moving the robot in linear and angular space.

Wall follow algorithm in maze solving

I have used the slightly modified wall follow algorithm to solve a simple connected maze. This wall-follow algorithm prioritizes the right-hand rule which means it will always keep the wall at its right side. The working principle is as follows:

i. It starts with moving forward until it gets a wall.
ii. Then it keeps the wall at its right side and keep moving forward.
iii. If it loses the wall, then it always turns right to get the wall again.
iv. If the front side is blocked, then it moves such a way that it gets the wall on its right.
v. Finally, it stops when it gets out of the maze in indicates that its sensor that's infinity readings.

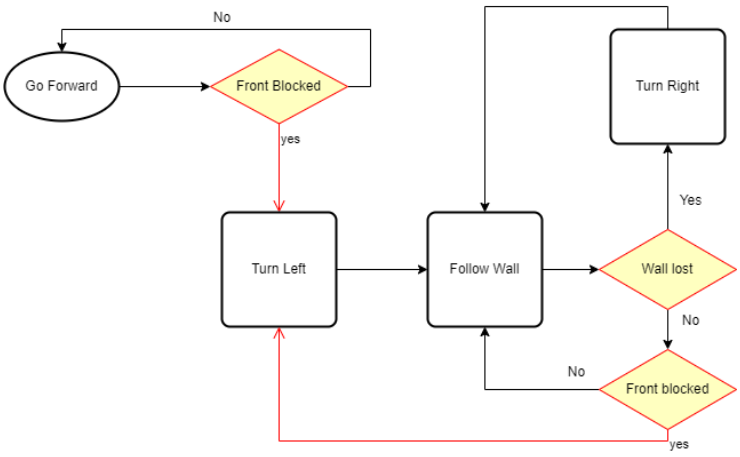


Figure 8: Wall follow algorithm prioritizing right hand rule

For this part of the task, I have used the same configuration as the wall follow node. I have just taken the minimum laser reading from the laser data staring

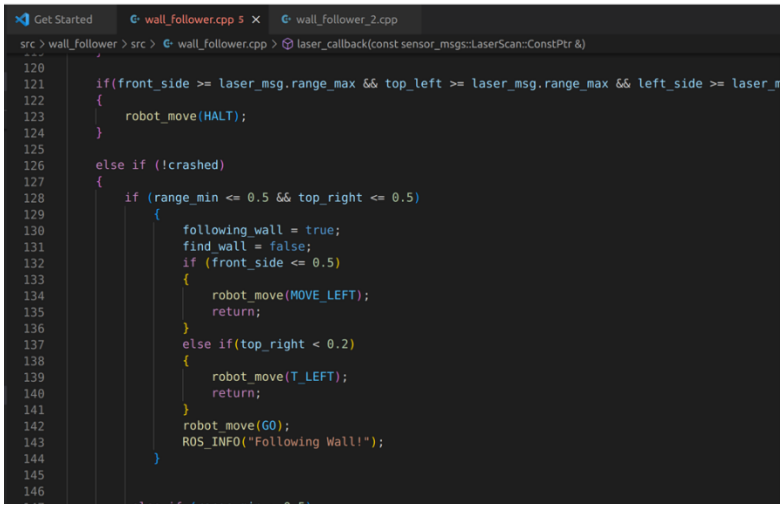


Figure 9: Implemented Wall-Follow for Maze Solving using C++

from angle 0 to 89 degrees which is right side to front side in terms of direction. Taking the minimum helped me overall the turning issue faced at the edges of the wall. This modified algorithm always focuses on wall on the right side of the robot.

Building Maze and spawning the robot

I have built a maze in the Gazebo simulator using model editor. I have designed the maze in 2D in the model editor which was made 3D by the gazebo simulator. Then I have added the wall design to the wall which is wood texture. I have saved the whole design as (.world) extension from gazebo simulator. For spawning the robot, I have created a launch file which specified the world I am using and the robot model. The launch file also includes the position of the robot where it should be spawned. With these steps I have made the maze for my maze solving task.

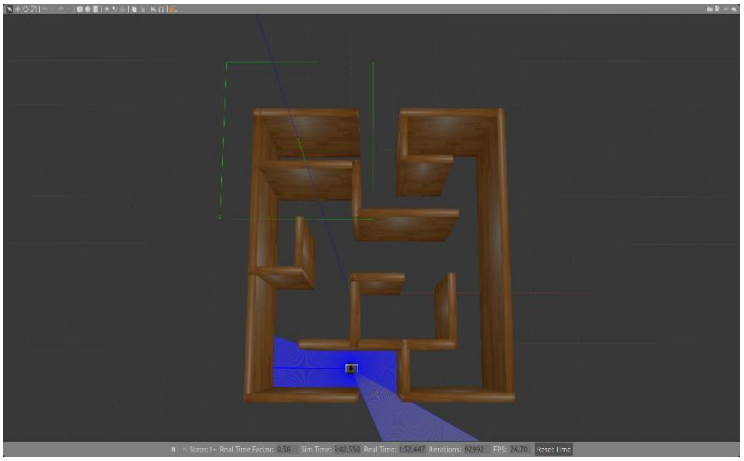


Figure 10: Simple Connected Maze

Demonstration

The following part contains the demonstration part of the implementation in form of screenshots. For the simulation of the implemented algorithm, I have used a software named "Gazebo Simulator". Gazebo Simulator is an open-source 3D robot simulator [2]. The figure 6 shows a Gopigo3 robot following a wall in house simulated in gazebo simulator.

Figure 11: Gopigo3 Robot Following a Wall in a House

The figure 7 shows a Gopigo3 robot model solving a simple connected maze with the wall-follow algorithm.

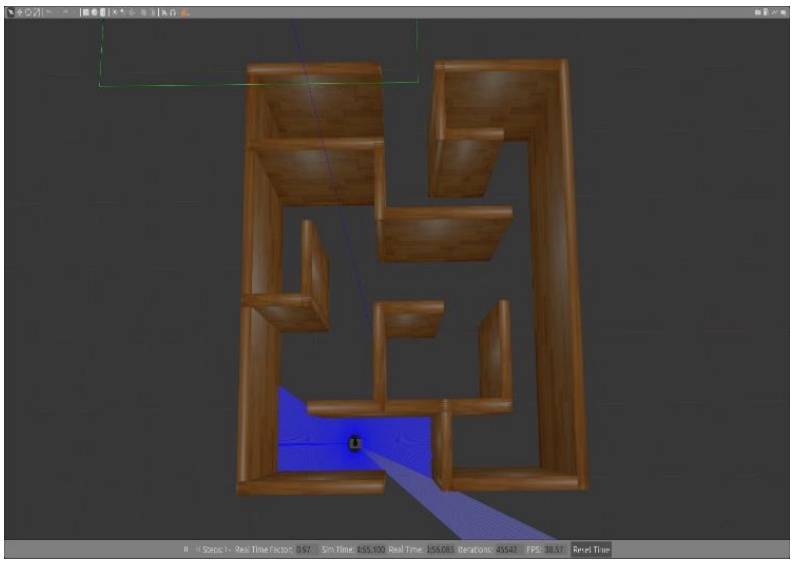


Figure 12: Wall following robot -1

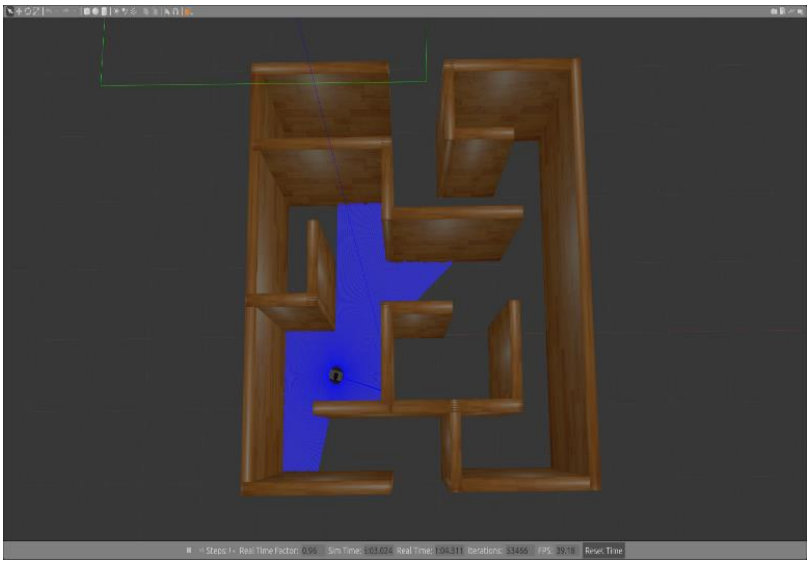


Figure 13: Wall following robot -2

Results

The simple wall-follow algorithm and the wall-follow algorithm for maze solving worked quite well. The simple wall-follow algorithm could follow the wall perfectly in a house made by us. The wall follow algorithm for the maze solving also could solve the maze following the right wall. The tasks were completed with quite reasonable results.

Figure 14: Visualized path on Rviz by wall follow algorithm

Learnings from the tasks

I have learned following things from the task:

a) Writing a ROS node using C++ language.
b) Wall-follow algorithm for robots.
c) Different maze solving algorithms available till now.
d) Using Gazebo Simulator to simulate the behavior of a robot.
e) Publishing and subscribing through a ROS node to a ROS topic.

Virtual Force Field Algorithm(VFF)

- ## What is a Obstacle Avoidance Algorithm?

In mobile robot navigation, one of the key factors is obstacle avoidance. For any type of mobile robot some obstacle avoidance algorithm should be provided for it's seamless movement. Obstacle avoidance is the ability of determining a collision free path by using the previous analysis of the data sent by different sensors that makes the robot enable to process the information about where the obstacles are located. It can be done locally or globally or both. The local planning concept is about using real time sensor data to identify and avoid nearby obstacles whereas global planning select the most suitable path for a robot to follow according to a map of the environment.

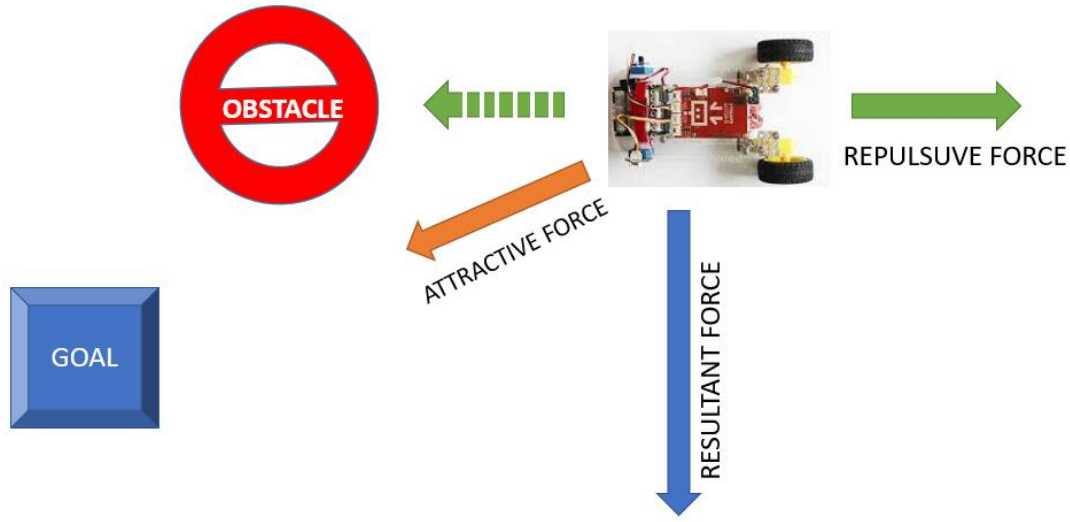
- ## Basic Concept of VFF

One of the popular obstacle avoidance algorithms is the VFF – Virtual Force Field. It is a real time obstacle avoidance algorithm and it is usually a local path planner. It follows the basics of the wall following techniques. Representation of obstacles with the certainty grid method was used in this algorithm. Though this concept is not used anymore, but it is important to know about this, because the modern form of the algorithm was created based on that. This certainty method was used because at that time we did not have modern sensors like lidar. The method was that each cell (i, j) contains a certainty value C(i, j) that indicates the measure of confidence that an obstacle exists within the cell area. The greater C(i, j), the greater the level of confidence that the cell is occupied by an obstacle.

Certain modifications were done by R.A. Brooks in this algorithm. According to Brooks' implementation each sensor range reading as a repulsive force vector. If the magnitude of the sum of the repulsive forces exceeds a certain threshold, the robot stops, turns into the direction of the resultant force vector, and moves on.

We can see the basic idea of the virtual force field in the picture below. It depends on the basic vector sum concept. The key is to calculating the forces. We have three forces in total, one is the Repulsive force, one is the attractive force and the final one is the sum of these two forces, the resultant force, in which direction the robot moves. There is an attractive force between the robot and the obstacle. Repulsive force vector is the vector which is same in the magnitude of the that vector but the direction is opposite. As a result, this vector force helps the robot car to move away from the obstacle. Now comes, the second vector, this is attractive vector force which works from the current position of the vector towards the final goal position. This vector force helps the car to move towards the final desired position. And the final resultant vector is the sum of this two vector in which direction the robot

moves.



Several key steps were followed to implement this algorithm, which are described below:

- **Understanding of goal pose (/move_base_simple/goal) and laser scan (/scan)**

Two of the main factors to implement this algorithm was to understand how to reach towards goal and managing the laser scan messages so that we can process them correctly in order to provide correct direction. By subscribing to these topics it was possible to do this tasks. Path planning consists of rotating in place to face the goal and then driving straight towards it. We can get a goal for move_basic to pursue via the simple interface. If data from a laser scanner is available, collision avoidance can be performed. If an obstacle is detected, it will slow or stop in an attempt to avoid a collision. We used some published topics here like cmd_vel which helps with velocity commands meant for execution by a mobile base. There are several parameters of the laser_scan like ranges which should be understood.

- **Transforming laser measurements to pointcloud in the robot frame (base_link)**

In order to get our laser scan as a set of 3D Cartesian (x,y,z) points, we should convert it to a point cloud message. The function we should use the majority of the time is the transformLaserScanToPointCloud function, which uses tf to transform our laser scan into a point cloud in another
frame. transformLaserScanToPointCloud uses tf to get a transform for the first and

last hits in the laser scan and then interpolates to get all the transforms in between and this is much more accurate.

- **Calculating of the forces and providing the velocity commands**

As discussed previously our main work is calculating the forces, the first thing we should calculate is the repulsive force. But before than we should understand the logic of its working. When the laser scan exerts its laser on the surrounding environment, there can be mainly two possibilities, one is that the laser distance reading is a specified magnitude, which means there is obstacle on that exerted laser part and another possibility is that the range of the laser in infinity, which means there is no obstacle. For the lasers that coincides with obstacle, we should store their value and sum them up to find the total repulsive force, which will help the robot to move from the obstacle amd towards the target goal point.

```
for(i=0; i<ranges;i++)
{

if( laser_ranges[i] < dangerous && laser_ranges[i] )
{
repForce= (dangerous - laser_ranges[i])*1.0/(dangerous - stopdis)
* (dangerous - laser_ranges[i])*1.0/(dangerous - stopdis) *
vect_max;
cnt++;
                        }
else if(laser_ranges[i] < stopdis )
{
      repForce= vect_infinit;
      cnt++;
}
else
{repForce x= 0;}
```

For the attractive force, from the current position of the robot we find the distance towards the goal point and calculate the force like below:

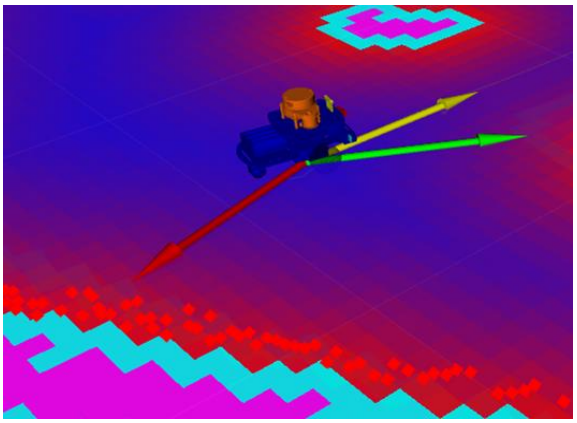
```
if (distance != 0){
 attractX = attractConst * ( x_d / distance);
 attractY = attractConst * ( y_d / distance);
 attractMag = hypot(attractX, attractY);
 attractAngle = atan2(attractY, attractX);
}
```

And lastly we had to calculate the resultant force which is just the addition of this two forces. After calculation of these forces we should provide proper velocity commands to move the robot towards the goal.

- **Defining a custom arrow marker in RViz for displaying forces**

As we have discussed before that we have three forces working on our mobile robot, so it would be very convenient to publish this forces as arrow marker using rviz, which will let us know that what the robot is thinking every moment, also is it

calculating the forces properly. For this we can simply use the rviz/DisplayTypes/Marker. It allows us programmatic addition of various primitive shapes to the 3D view by sending a visualization_msgs /Marker messasge. Our taks was to connect the forces with three different arrow markers, which can be seen in the below picture, that the yellow arrow points to the goal(the attractive force), the red one is the repulsive force, and the green one is the resultant force.



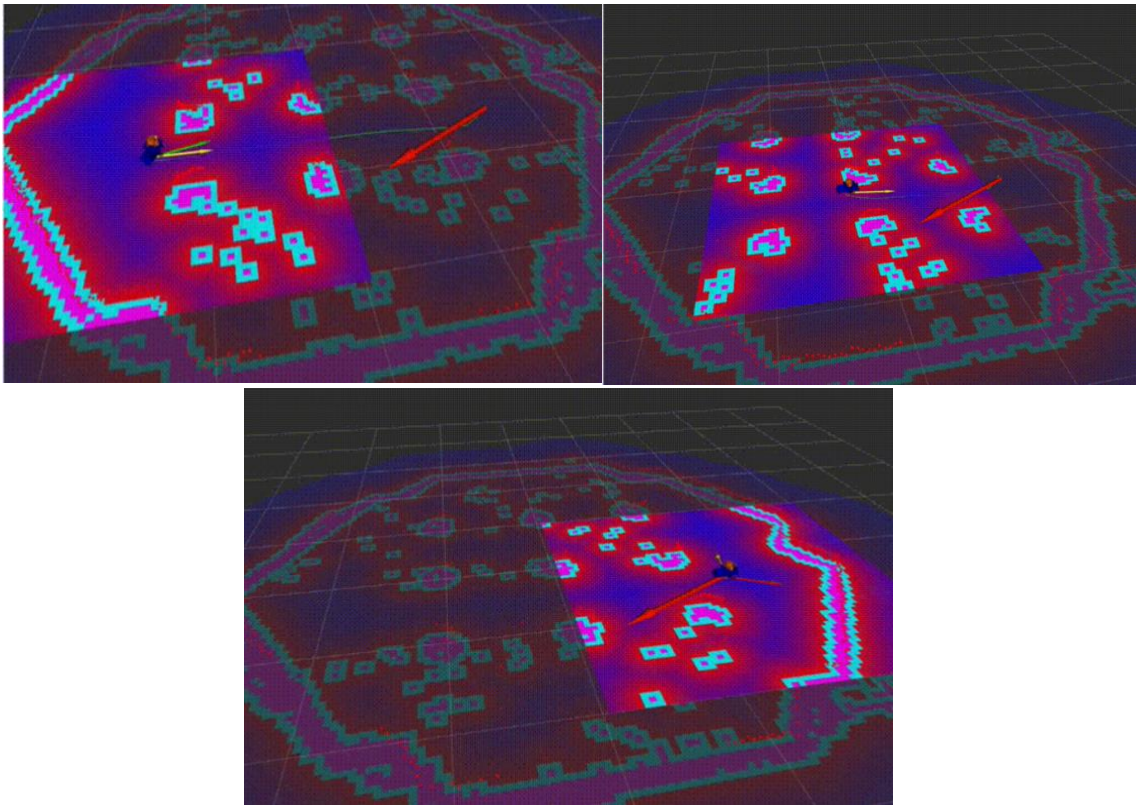
- **Implement VFF algorithm as a move_base local planner plugin**

Finally, we have implemented the vff algorithm as a move_base local planner plugin, to do so we created a new class and added a header file. Integration of the following functions were done which are necessary and common steps to add any local planner :

```
 void initialize(std::string name, tf2_ros::Buffer* tf,
  costmap_2d::Costmap2DROS* costmap_ros); //Constructs the ros
wrapper.

  void odomCallback(const nav_msgs::Odometry::ConstPtr& msg)

 bool computeVelocityCommands(geometry_msgs::Twist& cmd_vel);
//Given the current position, orientation, and velocity of the
robot, compute velocity commands to send to the base. //we assume
that the odometry is published in the frame of the base

 bool isGoalReached(); //Check if the goal pose has been achieved


 bool setPlan(const std::vector<geometry_msgs::PoseStamped>&
orig_global_plan); // Set the plan that the controller is
following
```

Besides, I created the code into several modules in order to construct and navigate easily. The modules were, vff.cpp, computeforce.h(for calculation of the virtual forces), marker.h(in order to publish the forces as arrow marker), vff_localplanner.h(contains all the functions and variables for the local planner plugin) and also the launch file for move base local planner , move_base.launch.

A demonstration of the algorithm can be seen below, where in the first picture the robot first determines it's angle and face towards the goal, in the second picture the robot is travelling avoiding the obstacles with the help of virtual forces and finally it reaches it's goal point :



- **Conclusion and future work:**

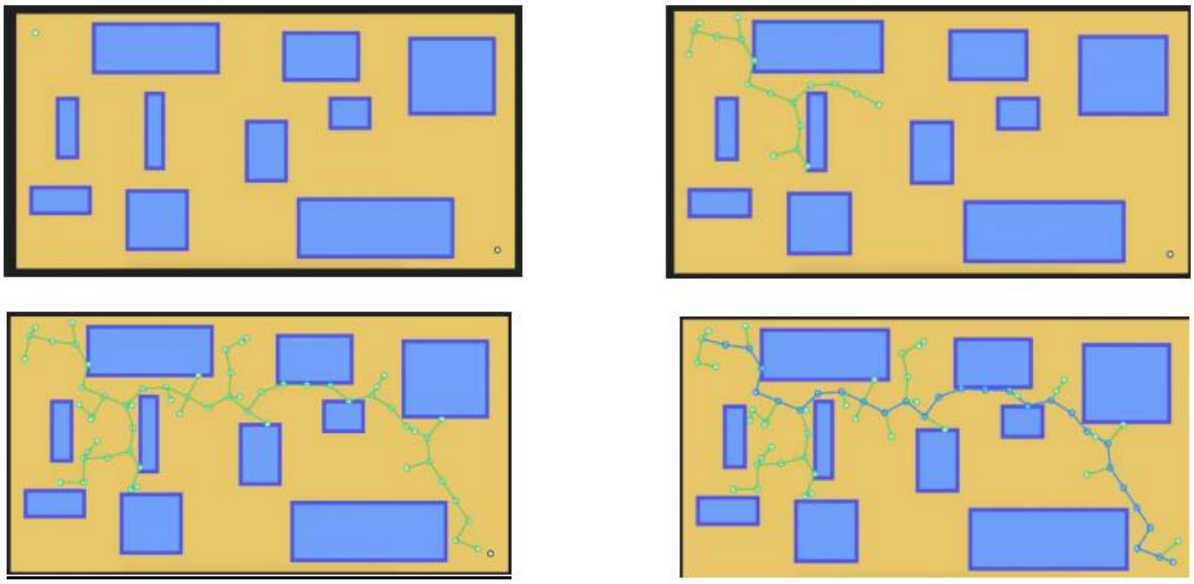
It can be said that VFF is a good and quite popular algorithm for obstacle avoidance. But there are certain drop backs of this algorithm. For example, it can be seen above that while it moves in a narrow corridor , where obstacle can be found on both sides frequently, it has to continuously change the direction, which makes the robot slower. Also sometimes when the path is really long and it has to make several round of the same obstacle it gets stuck, so some there are some error with the efficiency that can be improved. So, in future I would like to work more with this algorithm to make it more efficient, at the same time I will try to integrate this algorithm with other path planning algorithm in order to maintain a seamless movement for the mobile robot.

- **Rapidly-exploring Random Tree**

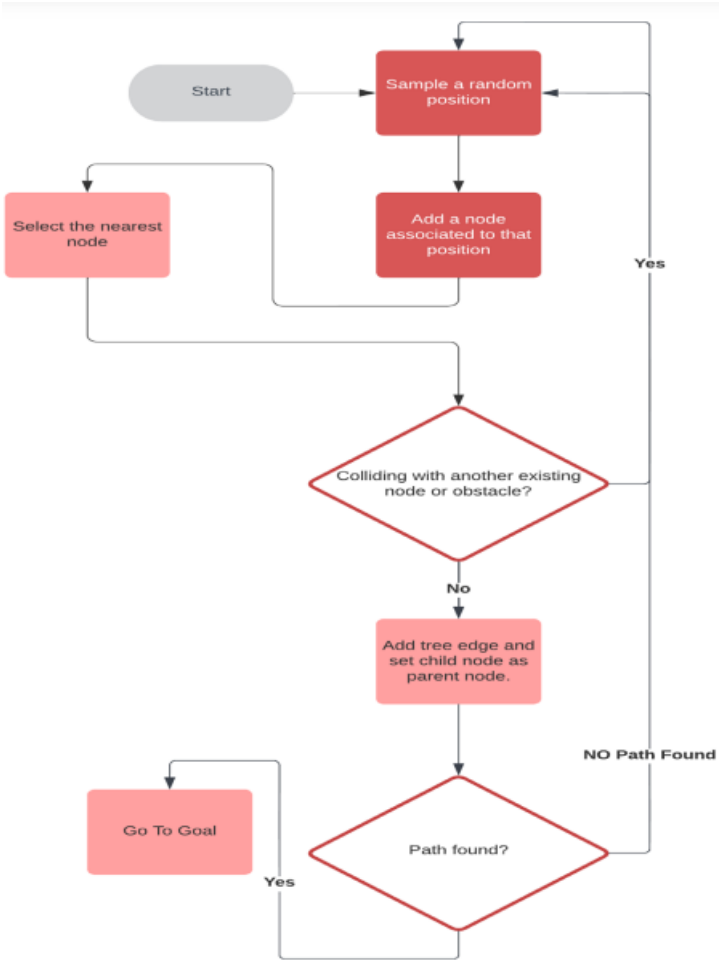
*) Problem Overlay: One really important part of a robot is its ability to plan a path from one location to another given a map of its environment. Whenever a robot has to go from a start location to a goal location to complete a task, it has to come up with a path plan for how to move around its environment. In the following we can see a concept of a path planning classical problem.

Here is a map and a robot is given a goal. The robot has to move through the map until it reaches the goal. That is called path planning. Now, one of the widely used and effective path planning algorithm is known as Rapidly-exploring Random Trees algorithm.

*) Rapidly-exploring Random Trees: In RRT, the robot has to find a path from the start to the goal using rapidly generated random trees. The tree should not be generated colliding with the obstacle of the world and find a path to the goal point.



*) Logic Behind Problem Solving: In this part we will discuss the logic that was used to solve this problem. As a first step we will initiate an empty tree. After that we will start adding nodes to the tree that represents the first node and starting position. We will keep repeating this step until we reach the goal.



Here we can see a Flow-Chart to see what was done in general to make the RRT problem possible to solve.

*) Programming Approach:

This approach is based on the flowchart we have created above. We will create 1 cpp and 4 header files to support the cpp file.
1) rrt.cpp: This is where the main code is written to create the rrt tree, make plan and publish the plan for the robot to follow.
2) Random_Gen.hpp : This header file is used to generate random number for the nodes
3) Nodes_And_2D Space: This header file will be responsible to find the nearest nodes, nearest edge, calculating distance between nodes etc.
4) rrt.hpp : This header file is created to generate rrt tree, calculating global path and so on 5) viz.hpp : This file is created to initialize and publish marker for the rrt tree generation.

Several local ROS header files were also used to ensure proper workflow for the algorithm to run. They can be found below:

*) <nav_core/base_global_planner.h> : Provides an interface for global planners used in navigation. All global planners written as plugins for the navigation stack must adhere to this interface

*) <tf2_geometry_msgs/tf2_geometry_msgs.h> : To use tf geometry messages

*) <geometry_msgs/Point.h> : To work with points in algorithm

*) <costmap_2d/costmap_2d_ros.h> : Working with cocstmap

*) <ros/ros.h> : Finally the ros header file which most generally includes publisher, subscriber, services, nodes, master, topics, name, params etc header files which are used frequently throughout a general ROS operation in ubuntu.

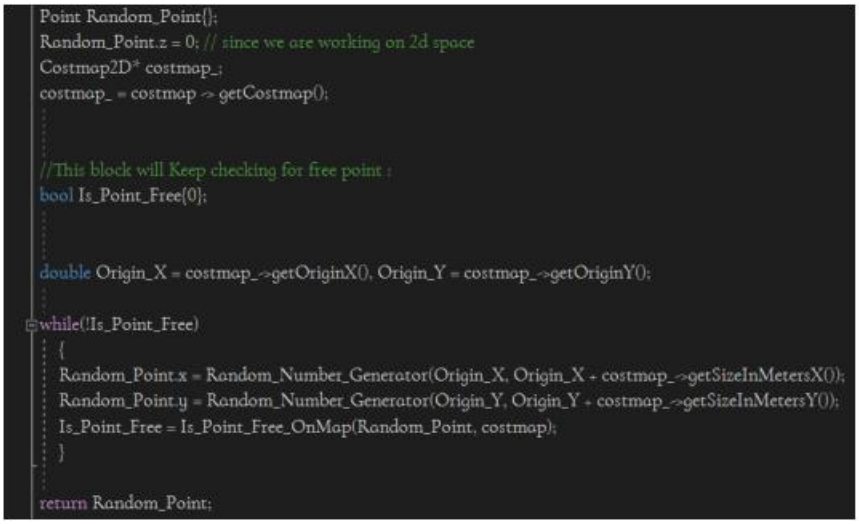
*) <visualization_msgs/Marker.h> : To use markers in ROS

*) <std_msgs/ColorRGBA.h> : used side by side with the previous header files, adding colors to various objects in the ROS system.

*) <geometry_msgs/PoseStamped.h> : This header file is also crucial to marker publishing and various other geometry messages.

In the following we will discuss some of the basic concepts that has been used in the algorithm so far. Firstly, we will generate a random point in the map. For this purpose, we have used

1) Point Get_Random_Point(Costmap2DROS* costmap) function. The purpose of this function is to find a random point which is free on the costmap and doesn't collide with other obstacle or any existing point and returns the point for further use. We can see the implementation below



2) We have also used Tree_Nodes Find_Nearest_Node(const Point point, const rrt *Tree) function to find the nearest node in the space to our parent node. In this function it finds all the nodes and calculates the

distance between them and if the distance is shorter, it will find the shortest distance node and expand in that direction. Implementation can be found below:

```
Point Nearest_Node_loc{};
Tree_Nodes Nearest_Node{};
int parent_loc{};
double Nearest_Distance{}, Current_Distance{};


// Iterating through the Nodes:

for(int k=0; k< Tree->Nodes.size(); k++)
  {
    //Assuring not the same node
    if(point.x != Tree->Nodes.at(k).vertex.x && point.y != Tree->Nodes.at(k).vertex.y)
      {
        Current_Distance = Find_Distance(point, Tree->Nodes.at(k).vertex);
        if(Current_Distance < Nearest_Distance)
          {

            Nearest_Distance = Current_Distance;
            Nearest_Node_loc = Tree->Nodes.at(k).vertex;
            parent_loc = k;
          }
      }
  }


 Nearest_Node.vertex = Nearest_Node_loc;
 Nearest_Node.vertex.z = 0;
 Nearest_Node.parent = parent_loc;

return Nearest_Node;
```

3) After this when a node is generated and selected we have used Tree_Nodes Extend_Node(const Tree_Nodes point_start, Point point_end, const double dist) function to extend the tree edge towards that node.

4) We have implemented as well the rrt Generate_RRT_Tree(PoseStamped x_start, PoseStamped x_end, Costmap2DROS* costmap, double tolerance, int Node_Num, double dist) function to generate a rrt tree of the global path that will be calculated by the algorithm.

5) A Function bool Global_Path_Calculation(“Params”) was also created to calculate the global path from the start to the goal.

There are other functions inside the header files that were also created to support the algorithm. These were one of the basic ones that were crucial to make the rrt algorithm work.

*) Adding as move_base plugin: Our original task is to add this RRT global path planner as a move_base plugin. To add this as a global planner plugin we have to export the class as a global plugin. On this note we have used this following command in the .cpp file to accomplish the task:

PLUGINLIB_EXPORT_CLASS(global_planner::RRT_Global_Planner,nav_core::BaseGlobalPlanne r)
Here we are using the nav_core's BaseGlobalPlanner and singing up out own own class global_planner:RRT_Global_Planner as a global planner plugin in move_base. Inside our class we will overewrite two crucial functions located inside the BaseGlobalPlanner class.

These two virtual functions are 1) void initialize() 2) bool makePlan()

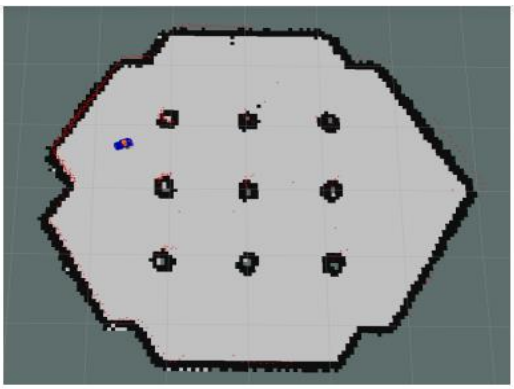
We will override these two functions with our own algorithm and implementation to make our algorithm to integrate with the global planner. Several YouTube videos were also followed in this task. Several other functions were also implemented inside our created class.

```
public:
    //Function For RRT Global Planner:

    RRT_Global_Planner();
    RRT_Global_Planner(std::string Name, Costmap2DROS* ROS_CostMap);
    ~RRT_Global_Planner(){}
    void initialize(std::string name, Costmap2DROS* costmap_ros);
    bool makePlan(const PoseStamped& start, const PoseStamped& goal, std::vector<PoseStamped>& plan);
    void publish_Plan(const std::vector<PoseStamped>& plan);
```

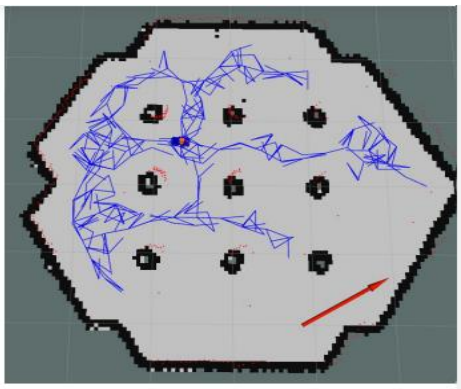
These classes are mostly constructor, destructors, virtual functions and lastly custom function publish_Plan() to publish the rrt plan for the robot.

Working Result:

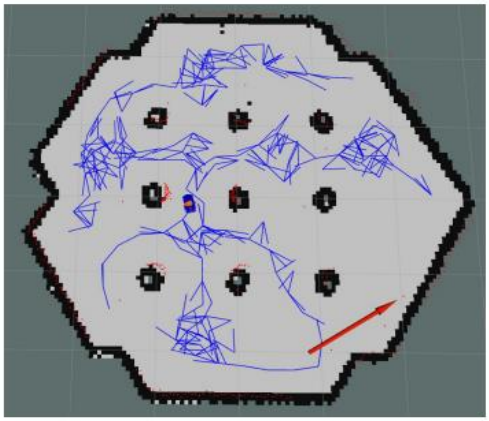
Below are some screen shots of a gopigo3 finding its way towards the goal.



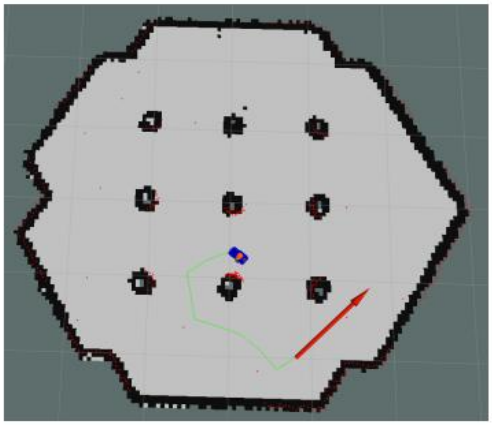
(1) Robot in its initial position



(2) Creating randomly generate tree to find a global path. Red arrow is the goal and the orientation of goal for the robot



(3)Robot approaching the target goal



(4) Tree is turned off, only global path shown which is derived from the tree

) Conclusion: Even though RRT planner is successful in finding the path from the start to the end point, but there is always a high possibility that the path found by this algorithm is not optimal. As a result, if too many resources are spent, optimal path planning will become a necessity for a ROS developer. On this note, a new algorithm has been developed based on the original methodology of RRT which is RRT (RRT star). Using this algorithm, a robot can find the most optimal path towards it goal.

References:

ROS wiki tutorial:
http://wiki.ros.org/navigation/Tutorials/Writing%20A%20Global%20Path%20Planner%20As%20Plugin%20in%20R OS ii A dummy global planner was followed to add this global planner as a plugin:
https://github.com/domikiss/dummy_local_planner iii
https://www.youtube.com/watch?v=We1gGDXAO_
Borenstein, Koren - 1989 - Real-time obstacle avoidance for fast mobile robots.pdf
Dummy local planner
https://github.com/domikiss/dummy_local_planner/blob/master/src/dummy_local_planner.cpp

"Maze-solving algorithm," Wikipedia. Apr. 20, 2022. Accessed: May 23, 2022. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Maze-solving_algorithm&oldid=1083691143

"Gazebo simulator," Wikipedia. Apr. 05, 2022. Accessed: May 23, 2022. [Online]. Available:
https://en.wikipedia.org/w/index.php?title=Gazebo_simulator&oldid=1081050879