TURTLEBOT NAVIGATION

For turtle-bot to start navigation from its initial position to the centre of the room (approx. position) first we launch the turtlebot navigation stack. For this first part the idea is to extract the location from the starting point to the centre of room, show in the figure below:

Figure 1: Fig showing initial starting position of the turtlebot

To execute the former task we run the navigation demo using our generated map file, following command is run following tutorials mentioned on ros wiki page, on turtlebot.

```
$ roslaunch turtlebot_navigation amcl_demo.launch map_file:=/tmp/my_map.yaml
```

This above command launches the amcl demo launch file. In amcl demo launch file four set of packages are initiated. The first one to get initialized is the 3dsensor. A 3dsensor launch file is launched which is responsible for activation of the 3d sensor rgb kinect mounted on the turtlebot in our case.

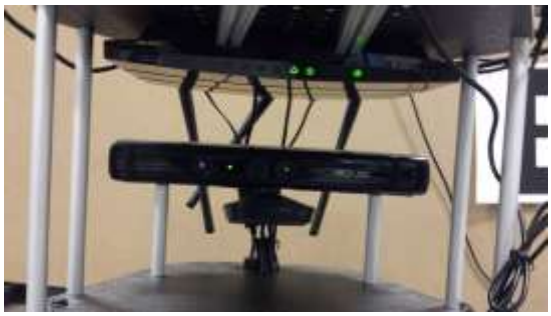


Figure 2: 3d sensor Kinect mounted on turtlebot

Second package is amcl demo launch file to get launched is the map server. This server provides the ROS node which offers map data as a ROS service. It also provides the command line utility which allows dynamically generated maps to be saved to file.

Following this, the AMCL demo launch file, initiates the 3d sensor amcl launch xml file. Where initial/ current position of the robot is equated to the zero and these initial pose parameters are passed to the 3d sensor (Kinect) amcl xml file.

Last node to be launched in Amcl demo launch file is the move base node. Move base node launches the move_base.launch.xml . This a ROS navigation stack with velocity smoother and safety controller. This move base node incorporates all the yaml files from turtlebot_navigation package stored in the param folder.

After executing the navigation demo command, on the workstation, following command is executed:

```
$ roslaunch turtlebot_rviz_launchers view_navigation.launch -screen
```

The former command bring the rviz screen that helps us visualize the turtlebot in our generated map. On rviz we use the initial pose command to illustrate our robots current position in the generated map. We navigate the turtlebot in the map and get the position of the following position.

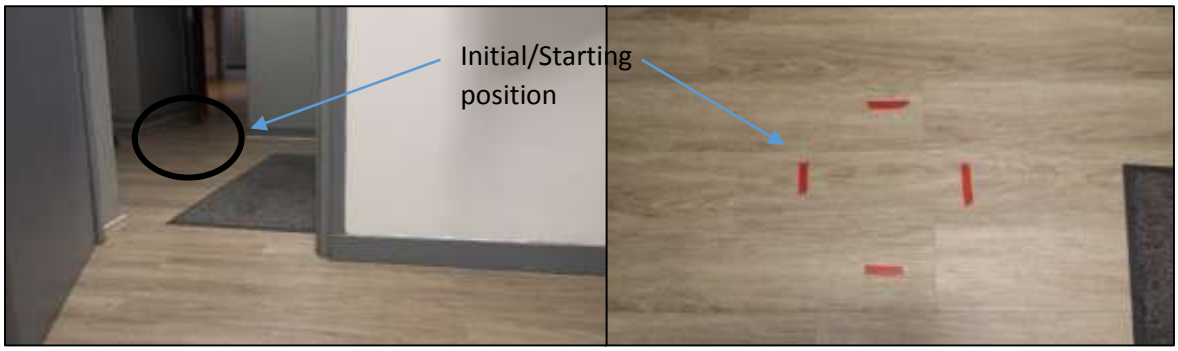
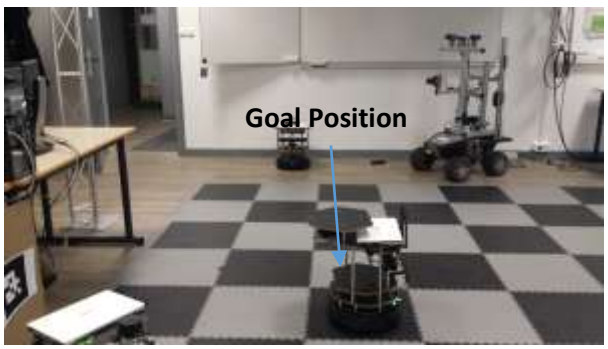


Figure 3: Fig showing starting position of turtlebot

We echo the turtlebot goal topic to print the current position/ pose of the turtlebot in the known (generated) map. We store these goal points as our initial/ starting points in the launch file as initial pose of the turtlebot. Using the similar procedure we get the pose of the robot when it reaches the centre of the room, this pose is saved as the goal pose.



Now once we have these start and goal pose values inside the launch file.
We simply just call this launch file. This launch file brings up the rviz, since we have already hardcoded our initial and goal position in the launch file so these points are already published on the turtlebot initial and goal pose topic.
So turtlebot starts navigation from the starting location towards its goal.

Ros Navigation

To perform ROS navigation we need two things:

- We need a map (Generated using the gmapping package)
- We need to be able to localize robot on that map.

Having a map alone without localization is useless, if the robot is unknown about its whereabouts. So in order to be able to achieve proper navigation task, robot should be able to know in which position of the Map it is located and with which orientation at every moment. This concept of knowing the whereabouts in the Map is referred to as 'Localization' in terms of ROS navigation. To achieve the locomotion of a robot in a known map, we need some sort of mechanism in ROS environment. This is achieved by using a Navigation stack. This stack is composed of set of ROS nodes and algorithms that are vital to move the robot from one point to another, while making sure obstacles are avoided that may arise in its(robot's) path. Navigation Stack, takes an input the current location of the robot, the goal pose, the Odometry data of Robot (wheel encoders etc..) and data from a sensor (in our case we used Laser sensor). An in return, it outputs the necessary velocity commands and forward them to the mobile base in order to move the robot to the designated goal, while making sure the turtlebot won't crash against obstacles, or get lost in the process. Following picture is taken from ROS wiki page that mentions the building blocks of Navigation Stack.

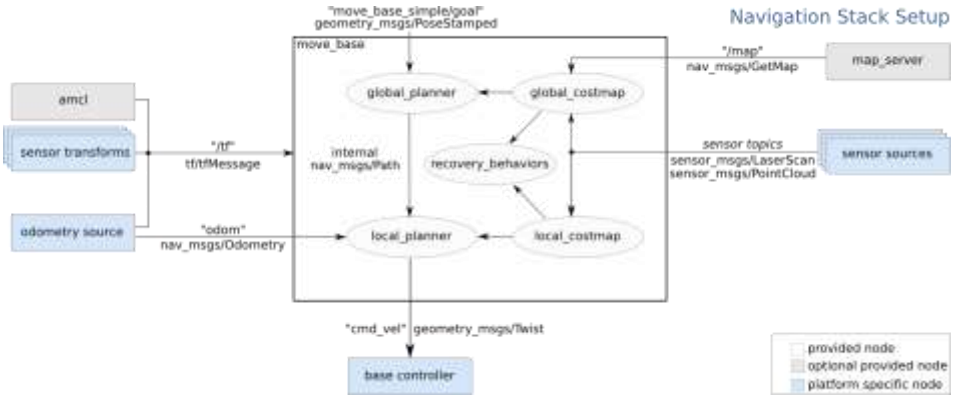


Figure 4: Navigation Stack

According to shown figure 4, some functional blocks must be presented in order to work and communicate with the Navigation Stack. Following are brief explanations of all blocks which need to be provided as an input to ROS Navigation stack.

- **Odometry source:** Odometry data of robot gives the robot position with respect to its starting position. Usually main odometry sources are wheel encoders etc. The odom value should publish to the Navigation stack, which has a message type of nav_msgs/ Odometry. The odom message can hold the position and the velocity of the robot.

- **Sensor source:** Sensors are used for two tasks in navigation: one for localizing the robot in the map (using for example the laser) and the other one to detect obstacles in the path of the robot (using the laser, sonars or point clouds).

- **Sensor transforms/tf:** the data captured by the different robot sensors must be referenced to a common frame of reference (usually the base_link) in order to be able to compare data coming from different sensors. The robot should publish the relationship between the main robot coordinate frame and the different sensors' frames using ROS transforms.

- **Base_controller:** The main function of the base controller is to convert the output of the Navigation stack, which is a Twist (geometry_msgs/Twist) message, into corresponding motor velocities for the robot.

Move_base node

This is the most important node of the Navigation Stack. The main function of move_base node is to mobilize a robot from its current position to a designated (goal) position. This node links the global planner and the local planner for path planning, connecting to rotate recovery package if the robot is stuck in some obstacle, and connecting global costmap and local costmap for getting the map of obstacles of the environment. So basically for navigation stack to work, we have move_base node, we need to provide it odom data, transforms of robot and laser data. Our robot should be localized in the give map. Following all this available info, we can send the goal pose to move_base node and move_base will give as an output proper velocity commands in order to move the robot towards the goal position as can be seen in the diagram 5.

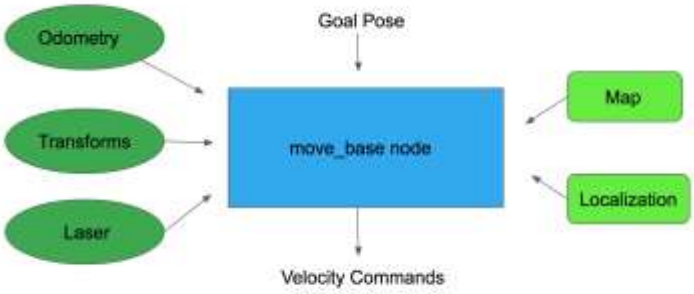


Figure 5: Move base node

Localization

When we initially start to move the robot in the generated map, it is unaware of its position in the map, so therefore the robot will not move as expected, it generates numerous guesses as to where it is going to move in the map. These guesses are known as particles (represented by tiny small green arrows when viewed in Rviz). Each particle contains the full description of future possible pose. When the robot observes the environment it is in via sensor readings, it discards the particles that do not match with these readings, and generates more particles close to those that look more likely. This way, most particles in the end cover the most probable pose the robot is in. So more the robot moves in the map, the more data its sensor will collect from surroundings hence the localization will be more precise. This is known as Monte Carlo Localization (MCL) algorithm, or also particle filter localization.

AMCL Package

The AMCL (Adaptive Monte Carlo Localization) package provides the amcl node, which practices the MCL algorithm in order to track the localization of a robot moving in 2D space. This node subscribes to the data of the laser, the laser based map, and the transformation of the robot, and publishes its estimated positions in the map. On startup, the amcl node initializes its particles filter. Previously we mentioned move_base is quite an essential node. This is due to the fact that main function of move_base is to move the robot from its current position to a goal position. Basically, this node is an implementation of a SimpleActionServer, which takes a goal pose with message type geometry_msgs/PoseStamped. Therefore we can send possible goals to this node by using a SimpleActionClient. This Action server provides the topic move_base/goal, which is the input of the Navigation Stack. This topic is then used to provide the goal pose. When this move_base node receives a goal pose on /move_base/goal topic, it links to components such as global planner, local planner, recovery behaviors, and costmaps, and creates an output, which is a velocity command with the message type geometry_msgs/Twist, and sends it to the velocity topic in order to move the robot.

Global Planner

When a new goal is received by the move_base node, this goal is immediately sent to the global planner. Then, the global planner is in charge of calculating a safe path in order to arrive at that goal pose. This path is calculated before the robot start moving, so it will not take into account the readings that the robot sensors are acquiring while moving. For calculating the path the global planner uses the costmap.

Costmap

A costmap is a map that represents places that are safe for the robot to be in a grid of cells. Usually, the values in the costmap are binary, representing either free space or places where the robot would be in collision. Each cell in a costmap has an integer value in the range (0-255). There are some special values frequently used in this range, which work as follows:

- 255(No_information): Reserved for cells where not enough information is known.
- 254 (lethal_Obstacle): Indicates that a collision- causing obstacle was sensed in this cell.
- 253(Inscribed_inflated_obstacle): Indicates no obstacle, but moving the centre of the robot to this location will result in a collision.
- 0 (Free_space): Cells where there are no obstacles and, therefore, moving the centre of the robot this position will not result in a collision. There exist two types of costmaps: global costmap and local costmap. The main difference between them is, basically the way they are built:

The globlal costmap is created from the static map. (The map generated using the gmapping package)

The local costmap is created from the robot's sensor readings. So the global planner uses the global costmap in order to calculate the path to follow.

Global Costmap

The global costmap is created from a user-generated static map. In this case the costmap is initialized to match the width, height, and obstacle information provided by the static map. This configuration is normally used in conjunction with a localization system, such as amcl.

Local Planner

Once the global planner has calculated the path to follow, this path is sent to local planner. The local planner, then, will execute each segment of the global plan. So given a plan to follow (provided by the global planner) and a map, the local planner will provide

velocity commands in order to move the robot. Unlike the global planner, the local planner monitors the odometer and the laser data, and chooses a collision-free local plan for the robot. So, the local planner can recomputed the robot's path while on move in order to keep the robot from colliding with any obstacles, yet still allowing it to reach its destination.

Local costmap

The first thing to remember here is that the local planner uses the local costmap in order to calculate the local plans. Unlike the global costmap, the local costmap is created directly from the robot's sensor readings. Given a width and height for the costmap, it keeps the robot in the center of the costmap as it moves throughout the environment, dropping obstacles information from the map as the robot moves.