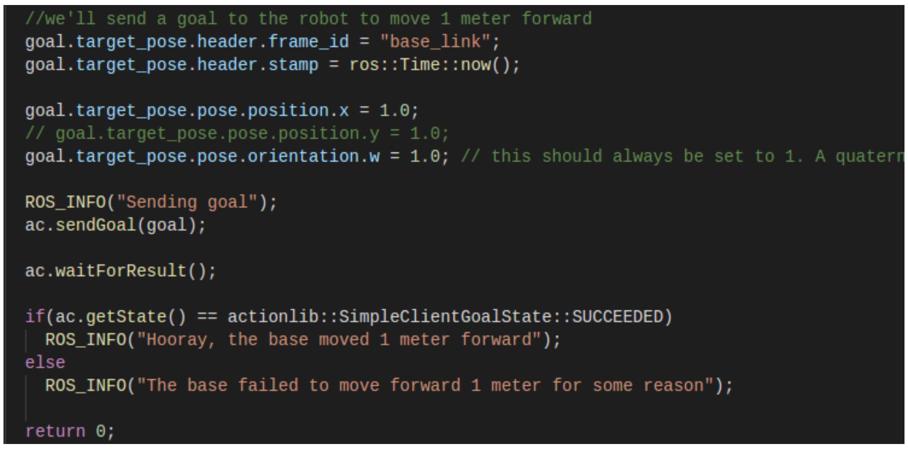
1 Planning

I am currently making a detailed monthly Gantt chart to describe all the remaining tasks of the Amcl localization method. This will be finished by the week-end and sent to you on the beginning of the following week. The aim is to better track the work‘s progress and set more stringent contraints. The file is in the form of .xls format.

2 Sending simple goals to the move_base node via actions

The first step in navigating the robot autonomously through a path is to send the **move_base** node **goals** via actions. The move_base acts as the server, while the actions as clients, so the move_base accepts all actions as clients. A code snippet of a custom made node called **simple_navigation_goals** is visible in Figure 1.

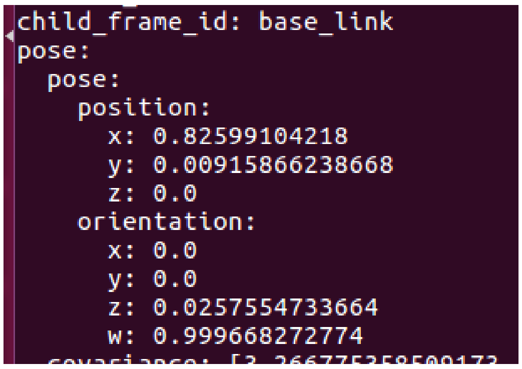


```
//we'll send a goal to the robot to move 1 meter forward
goal.target_pose.header.frame_id = "base_link";
goal.target_pose.header.stamp = ros::Time::now();

goal.target_pose.pose.position.x = 1.0;
// goal.target_pose.pose.position.y = 1.0;
goal.target_pose.pose.orientation.w = 1.0; // this should always be set to 1. A quatern

ROS_INFO("Sending goal");
ac.sendGoal(goal);

ac.waitForResult();

if(ac.getState() == actionlib::SimpleClientGoalState::SUCCEEDED)
  ROS_INFO("Hooray, the base moved 1 meter forward");
else
  ROS_INFO("The base failed to move forward 1 meter for some reason");

return 0;
```

Figure 1: Implementation of an action to send the robot a goal

The **goal** object (not visible in the Figure 1) is of type **nav_msgs::MoveBaseGoal**, which is the data type used for the **move_base** to send data. For this specific case I tested the robot to move 1 meter forward, hence the **x** is set to one. By editing properly the CMakeList an executable is created, under the devel/lib folder, and by **rosrun** the node, the robot nicely moves one meter forward.

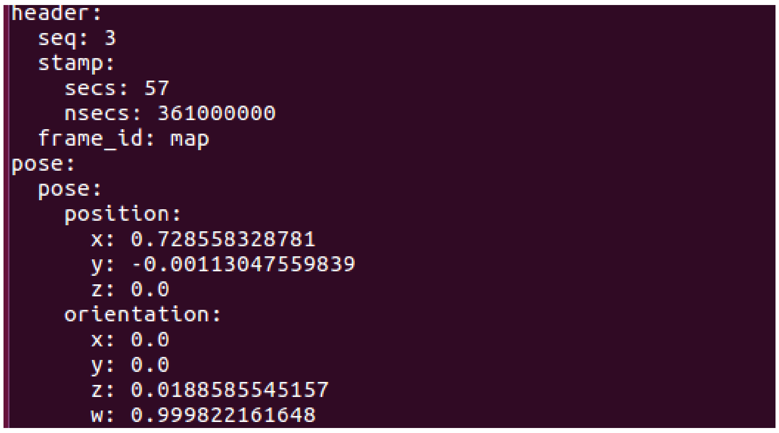
By echoing the **odometry/filtered** topic however, the results in the x-direction are different, as seen in Figure 2, and the same holds when echoing the **amcl_pose** topic, as seen in Figure 3.



```
child_frame_id: base_link
pose:
  pose:
    position:
      x: 0.82599104218
      y: 0.00915866238668
      z: 0.0
    orientation:
      x: 0.0
      y: 0.0
      z: 0.0257554733664
      w: 0.999668272774
```

Figure 2: odom filtered

For both cases the reduction is around **20 percent** less. I am not sure why now, but it is most probably due to parameters in the **global and local costmap** and **amcl.cpp**.



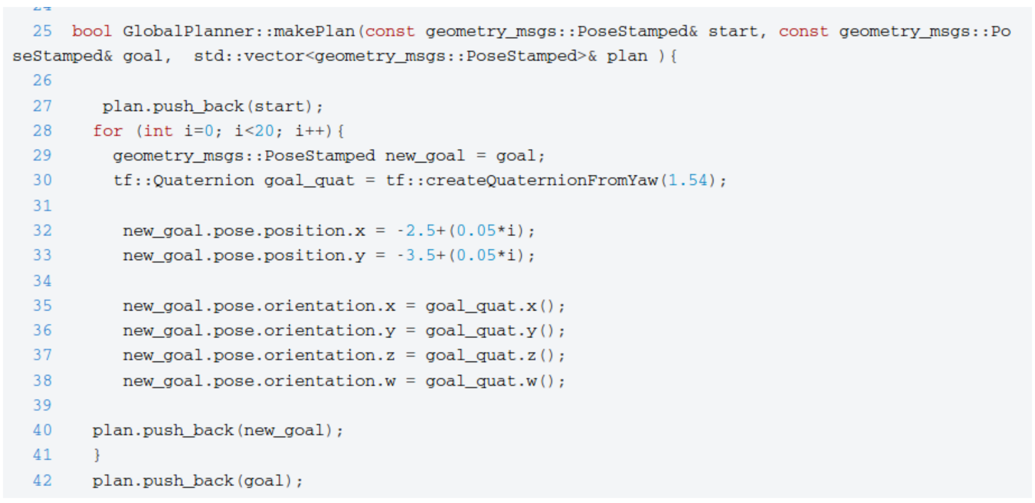
```
header:
  seq: 3
  stamp:
    secs: 57
    nsecs: 361000000
  frame_id: map
pose:
  pose:
    position:
      x: 0.728558328781
      y: -0.00113047559839
      z: 0.0
    orientation:
      x: 0.0
      y: 0.0
      z: 0.0188585545157
      w: 0.999822161648
```

Figure 3: amcl pose topic

3 The Global planner as a reference path

It is possible to define an own Global planner for the robot to follow, as a plugin. This is appended to the **move_base.launch** file as a parameter, and overrides the functionalities of the already implemented Global planner. What the Global Planner in Figure 4 does is to fill a vector of **plan** with dummy poses created in the for loop. In this way the robot, once the goal is set, as to follow the path. This however, does not move the robot in any direction autonomously. To do so, the **next to do** is couple the functionality of Section 2 and send an action for each new step goal reached. Currently I am figuring out to do couple those 2 node to work in **tandem**.

As input for the **new_goal**, instead of single dummy points, I will read coordinates from a .txt file, and save each line in a string. This has been already made in Matlab, and consists of 3 columns and N goals rows.



```
bool GlobalPlanner::makePlan(const geometry_msgs::PoseStamped& start, const geometry_msgs::PoseStamped& goal,  std::vector<geometry_msgs::PoseStamped>& plan ){

  plan.push_back(start);
 for (int i=0; i<20; i++){
   geometry_msgs::PoseStamped new_goal = goal;
   tf::Quaternion goal_quat = tf::createQuaternionFromYaw(1.54);

    new_goal.pose.position.x = -2.5+(0.05*i);
    new_goal.pose.position.y = -3.5+(0.05*i);

    new_goal.pose.orientation.x = goal_quat.x();
    new_goal.pose.orientation.y = goal_quat.y();
    new_goal.pose.orientation.z = goal_quat.z();
    new_goal.pose.orientation.w = goal_quat.w();

 plan.push_back(new_goal);
 }
 plan.push_back(goal);
```

Figure 4: The simple Global Planner as plugin