ROSbot Report

The purpose of this report is to give a personal reflection of the ROSbot project. ROSbot is an autonomous, open-source robot platform that runs the CORE2-ROS controller. As its name indicates, ROSbot is based on the Robot Operating System(ROS) and can be programmed either with c++ or with python. ROSbot comes with multiple sensors, Lidar, camera and many other electronic components. The purpose of this project is to program the ROSbot in order to move from an initial point to the finish line autonomously by avoiding to collide with obstacles. This project took place in a lab room. In order to simulate a natural environment for the ROSbot, multiple obstacles were placed around the place in order to detect the credibility and reliability of our code in many different obstacle scenarios. As such multiple small and big obstacles were placed in positions the robot would typically pass through. Next, will be presented the software architecture and the detailed software implementation. Based on the final code, several tests have taken place before the challenge. The results of the tests will also be presented in order to identify the capabilities and weaknesses of ROSbot.

Problem analysis

The systems design should be based on the subsumption architecture. In this way, there will be a hierarchy of the actions and behaviours of the robot. This hierarchy includes **obstacle avoidance** and **travelling to the finish line**. The **obstacle avoidance** is a crucial factor for the ROSbot to continue travelling around. It holds the lowest level in the hierarchy with the highest priority. The second level contains, **travelling to the finish line**. Higher layers in this hierarchy should never disturb the lower levels that have the priority. Only when a lower level is not active, a higher level can 'subsumes' control. The system was designed based on the fact that avoiding obstacles is of higher priority than navigating until the finishing point. In this way, ROSbot should concern about directing to finish line only when there is no near obstacle.

In order ROSbot to be capable to detect obstacles and handle the next actions all of the four robot's sensors have to be utilized. The distance sensors can detect object at a minimum distance of 3 cm and a maximum distance of approximately 0.9 meters. The two front distance sensors, right and left, can provide data of detected obstacles when ROSbot is moving forward. Depending on the feedback from the front distance sensors, the robot has to select the side that will pass the obstacle and customize its velocity accordingly. As such if only the right sensor detects obstacle then the robot should pass it from the left. Moreover, a small distance from the obstacle should lead the robot to make big turns, while big distances should lead to a small turning angle. Lastly, the rear distance sensors are valuable for the backward movements. Moving backwards is not a common situation, but it is needed in cases we are very close to an object or we have to escape from a trap.

ROSbot has to complete the race and direct to the finish line. For that purpose it should regularly check the direction and if needed to correct it. Leaving the ROSbot alone without navigation would lead to explore the surrounding environment by moving in random position depending on the obstacles. To prevent such a situation robot should detect which is the current angle and compare it with the initial angle. Prerequisite for this technique is the ROSbot to be placed with its front side to face the finish line.

Software Architecture

Next, it would be beneficial to present the detailed software architecture that was used for code development. ROS can be considered as a set of tools to create solutions. It provides drivers and implemented algorithms which are publicly available. For this project, we used **Nodes** and **Topics**. **Nodes** are in charge of handling devices and computing algorithms. Each of them handles a different task. **Topics** on the other side are used as data stream to exchange information between nodes. **Nodes** can publish to Topics and also subscribe. As such, in order to transfer a message from one node to another, the first node needs to publish the message in a given topic while the second node needs to subscribe to this topic. This technique can be summarized in figure 1 that describes now the topics and nodes communicate through publishing and subscribing to Topics. This figure illustrates also the Master which provides naming and registration services to the rest of the nodes in the ROS system. It tracks publisher and subscribers to a topic. Its role is to enable the nodes to locate one another.

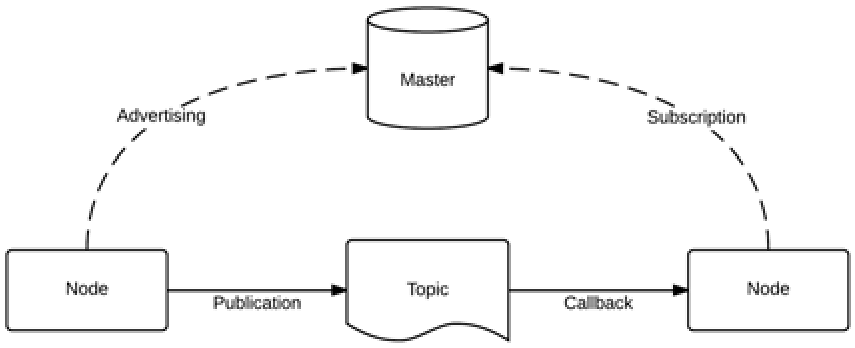


Figure 1: software architecture Nodes/Topics/Master

As previously described Topics are used as an intersection for the communication of two different nodes. Figure 2 illustrates the active nodes/topics of the code when the software is executed and it presents the connections between them. The ROS graph, that creates a dynamic graph of what is going on in the system, was extracted with the Linux command "**rqt_graph**". As can be seen from the following figure there are nine active nodes and eight active topics. For the purposes of the project, we designed four of the existing nodes. These are the "**action_controller**" , "**driver_controller_node**", "**imu_controller_node**", "**distance_node**". Next, we will describe the connection between nodes and topics in order to reach a high understanding of the implemented software architecture.

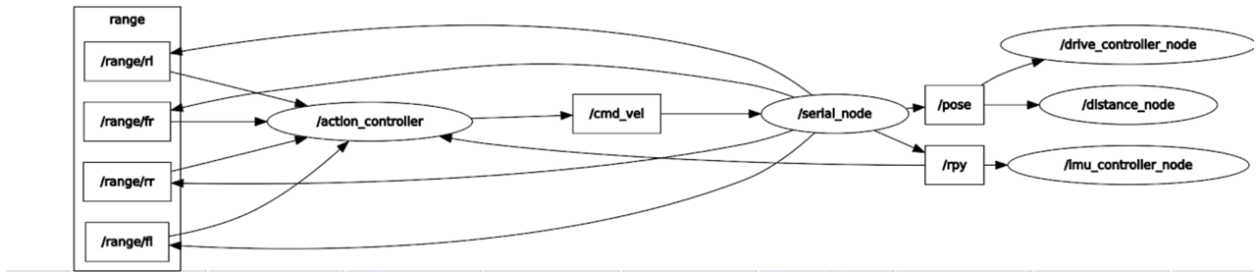


Figure 2: ROSbot ROS graph extracted from rqt_graph

According to figure 2, we can see that serial_node subscribes to most of the topics. Serial_node is a ROS node that can communicate with the CORE2 controller, through a serial link that connects the Husarion CORE2 controller board and the ASUS Tinkerboard. It should be mentioned that the /serial_node, that works as a bridge, is always activated by running the command " /opt/husarion/tools/rpi-linux/ros-core2-client /dev/ttyCORE2 ". The /serial_node publish to the /pose , /rpy and to /range topic.

The "action controller" node works as the brain of the system. This is where the decisions are made regarding the surrounding environment. For this purpose, action_controller node subscribes to four "range" topics. Each range topic reports the range to the nearest obstacle measured. As such "/range/rl" reports the distance feedback of the rear left sensor, while "/range/fr" the feedback of the front-rear sensor. The "/range/rr" and the "range/fl" report the distance feedbacks of the rear right and front left sensors respectively. All of these distances will be utilized in order to code robot to avoid colliding with an obstacle. Action controller subscribes also to the "/rpy" topic which provides the current orientation data of the robot coming from the IMU. More specifically,"/rpy" provides a message for the rotation in degrees, about the x,y and z-axes, regarding the position of the robot when it first switched on. Action can handle this information to restore and fix the current direction of the robot according to the initial direction. After decision making, action controller node publishes to "/cmd_vel" topic the linear and angular velocity components. In order these velocities to be transferred to the CORE2 controller, "/serial_node" subscribes to the "/cmd_vel" topic.

Distance_node is responsible for calculating the total distance the robot travelled from the initial position until the current position. For these calculations, there is a need for identifying the ROSbot position every some millisecond. The "/pose" topic can provide information for the position and the orientation of the robot. In the ROS graph, we can see this dependency through the distance_node which subscribe to the "/pose" topic where the current position of the robot is available to be retrieved.

For simulation purposes of the ROSbot, the RVIZ software has been used. RVIZ simulates the ROSbot movements in a specially designed environment and it provides useful feedback from the sensors. For the visualization purposes, two more nodes have been inserted to the package. The "imu_controller_node" read data from the ROSbot IMU by subscribing to the "/rpy " topic and generates a new frame relative to /base_link called /odometry. This allows visualization of the IMU data in Rviz. Similarly, the "drive_controller_node" read data published on the /pose topic and generate a new frame relative to /base_link called /odom. This allows visualization of the odometry data in Rviz.

Software implementation

All the c++ coding scripts have been stored in the directory called "tutorial_pkg". The project requires multiple files and c++ scripts to be executed. In order to avoid delays and launch several nodes and other processes in one command, a launch file was created. This launch files, called " group2_challenge.launch", includes all the necessary nodes created for the project as well as other important nodes and commands for the visualization of ROSbot to RVIZ. RVIZ commands have been deactivated during the challenge as can be seen in the ROS graph(figure 2). As previously mentioned there have been created four nodes, named action_controller, driver_controller_node, imu_controller_node, distance_node. It should be mentioned that one more node was created in order to facilitate the process of stopping the robot. It is called failsafe node. Next, we will describe the functionality of each node individually to create a better understanding of the code.

Action_controller is the most important node that deals with the decision making of the next ROSbot movement. In order to facilitate the development of the code, we utilized the hierarchical Finite State Machine(HFSM) in order to develop the action controller functionalities. In the main function, action_controller node is initialized and the loop rate is specified to 100ms. Also in the main function the initial speed values, linear and angular, were set to zero. In order ROSbot to sense the environment and detect possible obstacles, we subscribe to the available distance sensors topics. Consequently, four different callback functions have been implemented for the incoming sensors messages. Their task is only to put values in appropriate variables. That values represent the distance of each sensor with the nearest obstacle. In one of these callbacks, the distFL_callback, is stored the code for the next robot movement. The code placed only on the distFL_callback and not in every callback because we wanted the code to run only one time per 100ms. However, in this callback, we are taking into account all the range values from the other sensors callbacks. Depending on the distance from the recognized objects and their place in the environment, different speed and rotations applied to the robot. For this purpose, several cases have been included that utilize the distance sensors feedbacks and drive the robot accordingly.

In order to drive the ROSbot, the action_controller publish actions to "/cmd_vel". As such in the main function, we prepare the publisher for linear and angular velocity command and at the end of distFL_callback function, we publish velocity command message. "/cmd_vel" includes linear and angular parts. For ROSbot only the linear.x and angular.z can be used as this robot can rotate only on the z-axis and can direct forward or backwards, so only the x linear part is used. Finally, in the main function, we are subscribing to the "/rpy" topic. As such from the imu_callback function we can restore the current orientation of the ROSbot regarding the orientation of ROSbot in the initial position.

A closer look at the actions made could reveal that there are three fundamental cases.

- **Obstacle not found/out of range:** On this case no obstacle detected or the obstacle is far away (distance higher than 60cm)from the robot. The robot is free to move forward in high speed without any rotational movement if its direction doesn't differ more than 70^o degrees than the initial direction. In case its direction is more than 70^o or less than -70^o then a rotational movement changes the direction of the robot in order to minimize the deviation. In this way, the robot will constantly move forward without losing its direction.
- **Detected obstacle in range (20-60)cm**. In this case, robot has detected one or more obstacles in the range of 20 to 60cm. The detection could have happened from one or from both front sensors. The robot has two possible ways to pass the obstacle, right side or left side. The decision is made after comparing the distance of the left and right sensor from the obstacle. A threshold has been entered in order to exclude any possibility the robot to change constantly directions if both sensors detect at approximately the same distance an obstacle. Consequently, if both sensors detect obstacle but the right sensor is closer to it for more than a threshold, then robot will pass it from the lefts side. It happens the opposite if the left sensor is closer to the object. In case no sensor is closer to the obstacle for more than the given threshold then the robot will select the right direction.
- **Safe state:** In this case, the robot has detected an obstacle in distance less than 20cm from the right or from the left. In such a condition, if only one sensor detects an obstacle in distance lower than 20cm then the speed is decreased immediately and the decision is made regarding with which side is better to pass the obstacle. As such if the distance between the right sensor and obstacle is lower than the left side sensor and obstacle, the robot will move from the left side.
  If both front sensors detect obstacle in distance lower than 20cm then there are two possible scenarios. In order to select the scenario, robot has to check if back sensors detect any obstacle in a range lower than 20cm. If not then robot moves backwards until front sensors stop detecting obstacle in the range of 20cm. This scenario is called **Backward movement**. In this scenario, there is enough space to move backwards and re-plan how to escape. The second scenario, which is called **blocked**, happens when even one of the back sensors detect an obstacle in a range lower than 20cm. In this case, robot is blocked and stops moving.

Figure 3 illustrates the flow graph of the action controller for decision making.
Before proceeding we should define the variables that are used in the flow graph.

- **distFL**: Distance between the front left sensor and the obstacle.
- **distFR**: Distance between the front right sensor and the obstacle.
- **distRL**: Distance between the rear left sensor and the obstacle.
- **distRR**: Distance between the rear right sensor and the obstacle.
- **cur_dir**: Current direction of the robot. Value from "rpy" topic
- **init_dir**: Initial direction of the robot. Value from "rpy" topic

As previously described actions are made regarding the sensors feedback. In order to control the ROSbot we need to publish to /cmd_vel topic a linear and angular velocity. The actions can include forward movement, backward movement and a rotation at the right or at the left. The forward or backward movements are performed by adding value to the linear.x velocity while orientation is been performed by adding value to the angular.z velocity. Higher values for linear.x means higher velocity for ROSbot, while higher value for angular.z means rotation with an angle of high degree. The sign of the value indicates the direction of the movement and the direction of the rotation, respectively.

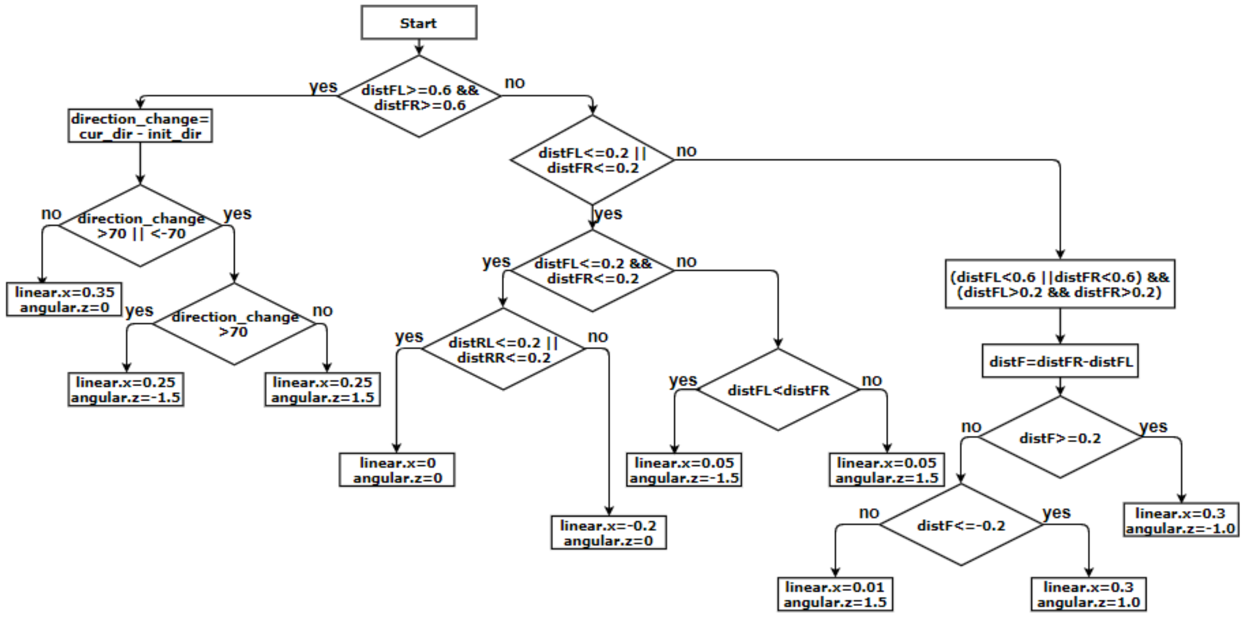

Start
distFL>=0.6 && distFR>=0.6
yes
no
direction_change= cur_dir - init_dir
direction_change >70 || <-70
no
yes
linear.x=0.35 angular.z=0
direction_change >70
yes
no
linear.x=0.25 angular.z=-1.5
linear.x=0.25 angular.z=1.5
distFL<=0.2 || distFR<=0.2
no
yes
distFL<=0.2 && distFR<=0.2
yes
no
distRL<=0.2 || distRR<=0.2
yes
no
linear.x=0 angular.z=0
linear.x=-0.2 angular.z=0
distFL<distFR
yes
no
linear.x=0.05 angular.z=-1.5
linear.x=0.05 angular.z=1.5
(distFL<0.6 ||distFR<0.6) && (distFL>0.2 && distFR>0.2)
distF=distFR-distFL
distF>=0.2
no
yes
linear.x=0.3 angular.z=-1.0
distF<=-0.2
no
yes
linear.x=0.01 angular.z=1.5
linear.x=0.3 angular.z=1.0

Figure 3: Flow graph of the action controller

Next, a distance node has been created to calculate the distance between the current and the initial position. In the main function, we initialize the distance node and specify its loop rate to 500ms. As such the distance calculation will be executed in every half a second. The current position of ROSbot is retrieved by subscribing to "/pose" topic in the main function. In the distance callback function, we retrieve the current position (x,y) and we call the calculate_Distance() function. The calculate_Distance() function calculates the distance between the previous and the current position with the distance formula (figure 4). The current position (x,y) is stored in two variables, marked as previous position, in order to be restored in the next iteration and perform the distance calculation again. This distance is added to the total distance variable which indicated the total distance the robot travelled from the initial position.

$$d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

Figure 4: Distance formula

Finally, the failsafe node is used as a convenient way to stop the ROSbot from moving around the place. ROSbot processes the code internally in the CORE2 controller after it is upload from a PC. As such whenever it is left alone to navigate the place it will continue to do it until it is left out of energy. In order to prevent such a situation, the failsafe node publishes to /cmd_vel zero values for linear and angular velocity, driving it to stop and rest till a new code is provided.

Tests

In order to test the credibility of our code and detect the performance of ROSbot, we assembled the robot reaction into multiple scenarios. Every scenario was performed with the robot placed in the ground while a computer was connected to the robot through a Wi-Fi connection. In this way, we could monitor the robot's decision making and detect possible bugs. In order to activate the robot, the " group2_challenge.launch" file was launched. Figure 7 illustrates some of the objects that were used as an obstacle to test the ROSbot.

At the first scenario, a cuboid object placed in front of ROSbot at a distance of approximately one meter away. Through the remote access, we launch the " group2_challenge. launch" file and the robot started to direct toward the object in a straight line. When it reached the distance of 60cm away from the object it rotated in a small angle while moving forward. After passing the obstacle it continued straight without any rotation. The next scenario was more demanding and included two objects (triangular and cylinder). Obstacles placed as like in figure 5. When the robot came closer to the first in row obstacle, it started rotating at the right. Next, the detection of the second obstacle, made robot to turn left and continue in a straight line.

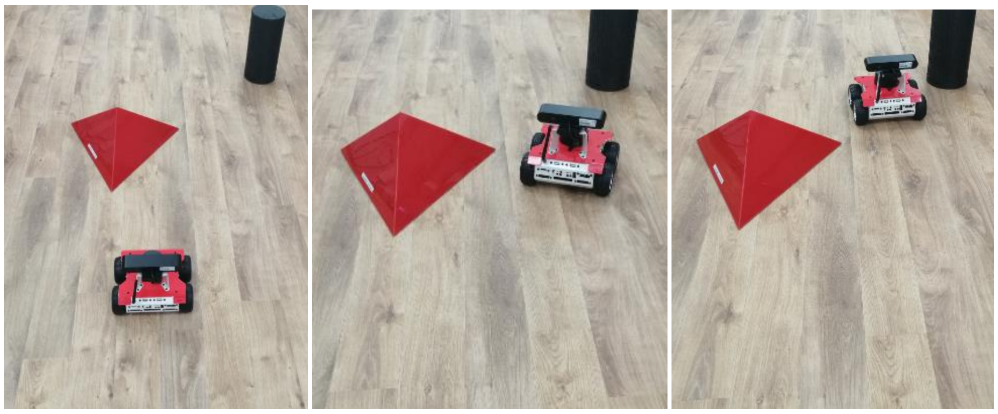


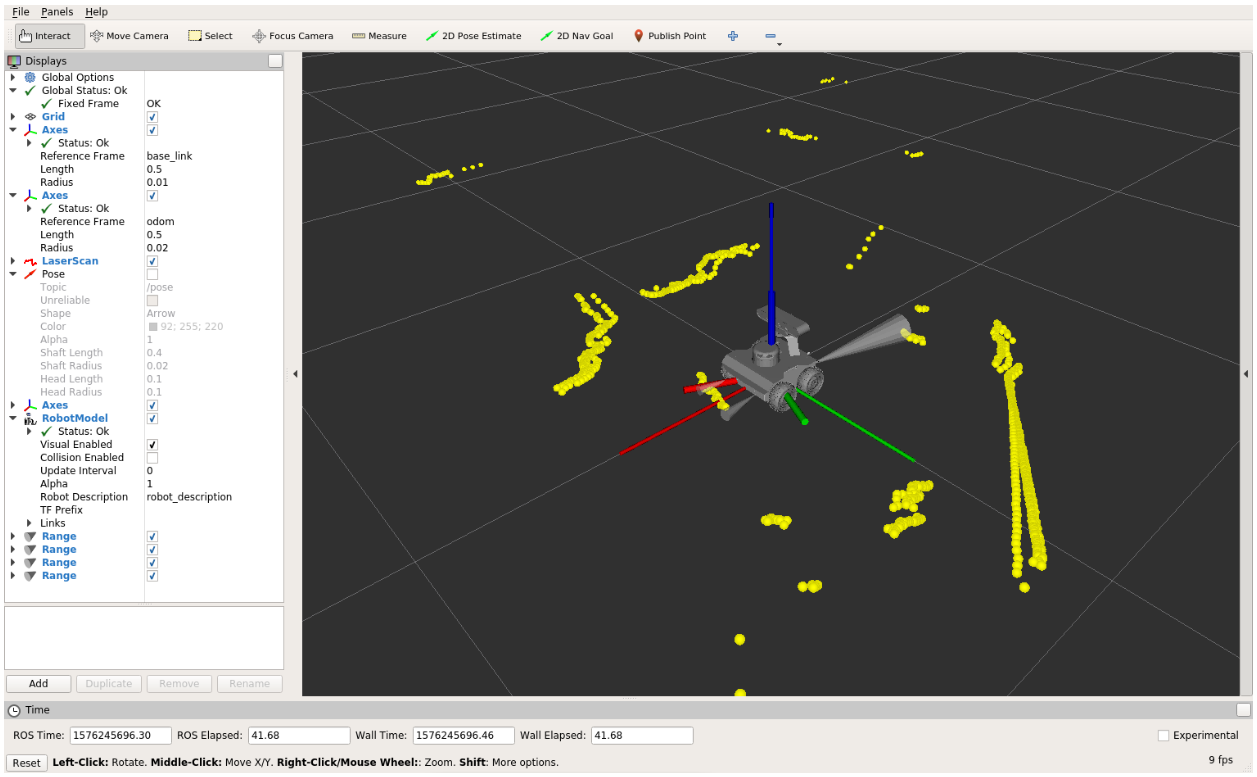
Figure 5:ROSbot passing obstacles (these photos are not the original but only a representation of the movements made)

The third scenario included the specially designed glass obstacle which includes a hole in the centre. Robot placed again in distance from the obstacle facing the centre of the obstacle. After travelling a small distance, the robot started to have fluctuations in its rotation. As such while it was heading toward the obstacle it was constantly making small rotations from the right to the left and the opposite. However, these small changes in the direction didn't affect the total direction of the robot, and finally, it passed the obstacle.

For the fourth scenario, we used the U-shaped obstacle, which works as a trap for the robot. We placed the robot in front of the obstacle and monitored its' reaction. The robot entered the trap and was directed towards the left wall. When it was close enough it rotated at the right to the forefront wall. In the forefront wall, the robot was rotating constantly from the right to the left and the opposite, without making forward movements. In case the robot happened to be very close to the wall, it started to go backward for a small distance. After it, it again entered the trap and repeated the previous problematic functionality. ROSbot didn't manage to escape from the trap.

Tests have been made for detecting its response when obstacles appeared in close distance all of a sudden. In these cases, the robot stopped and moved backwards. After this robot successfully passed the obstacle, sometimes from the left and sometimes from the right. Furthermore, robot was checked if it could detect obstacles that are some millimetres above the surface. When obstacles were a bit higher than the distance sensors height, the robot couldn't detect the obstacle. Finally, unsuccessful was the detection of an obstacle when the last was very small in size. For example, the robot couldn't detect the chair's legs.

During the development and testing of the robot we used several ROS tools in order to reach a high understanding of the system and to correct the malfunctions. "rostopic echo" is a tool that shows what is being published in a topic in a certain time. In order to test the ROSbot sensors and identify if they respond correctly to an obstacle depending on the distance, we run the command "rostopic echo /range/fl" and "rostopic echo /cmd_vel" while robot was placed in a stable position. By making gestures in the front left sensor we could see in the screen the distance between the hand and the sensor. At the same time, from the second rostopic we could monitor if the publishing values in the cmd_vel for angular and linear velocity were according to our code. The echo tool was used also for recognizing the positive and negative rotation of ROSbot. By rotating the robot in the z-axis and monitoring the change in z orientation ("rostopic echo /pose") we could identify the sign of each rotation. The testing included close observation of the active nodes, topics and their connections, while the RVIZ was also used as a testing simulation to identify the sensors and Lidar feedback according to the surrounding environment (Figure 6).



```
Global Options
Global Status: Ok
  Fixed Frame            OK
Grid                     ✓
Axes                     ✓
  Status: Ok
  Reference Frame        base_link
  Length                 0.5
  Radius                 0.01
Axes                     ✓
  Status: Ok
  Reference Frame        odom
  Length                 0.5
  Radius                 0.02
LaserScan                ✓
Pose
  Topic                  /pose
  Unreliable
  Shape                  Arrow
  Color                  92; 255; 220
  Alpha                  1
  Shaft Length           0.4
  Shaft Radius           0.02
  Head Length            0.1
  Head Radius            0.1
Axes                     ✓
RobotModel               ✓
  Status: Ok
  Visual Enabled         ✓
  Collision Enabled
  Update Interval        0
  Alpha                  1
  Robot Description      robot_description
  TF Prefix
  Links
Range                    ✓
Range                    ✓
Range                    ✓
Range                    ✓

ROS Time: 1576245696.30   ROS Elapsed: 41.68   Wall Time: 1576245696.46   Wall Elapsed: 41.68
Left-Click: Rotate. Middle-Click: Move X/Y. Right-Click/Mouse Wheel:: Zoom. Shift: More options.   9 fps
```

Figure 6: Simulation of ROSbot with RVIZ. The front left sensor detects a close obstacle. The rear left sensor detects an obstacle in a big distance(approximately 70cm).

It should be mentioned that for the final code there was a check on the CPU usage while the code was running. This test was performed with the "top" command. This test could reassure that the CPU was running at normal levels and could handle the multiple messages that were exchanged between nodes. This test was based on previous findings that when the loop rate of the action controller was low and simultaneously we were running the RVIZ simulation software, the sensors of the robot stopped working and the CPU usage exceeded the 100%. Furthermore, the test included the testing of the ROS graph in order to specify if the connections between the nodes and topics were correct.

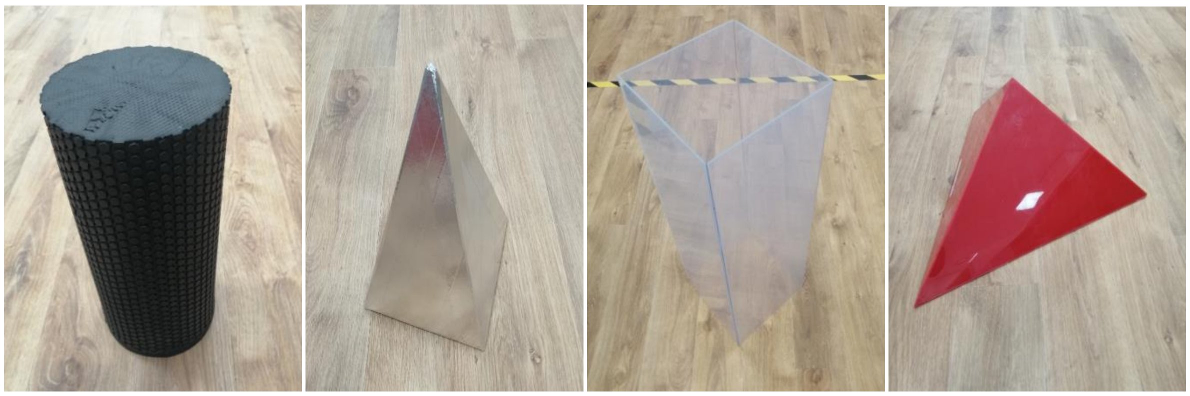


Figure 7: Obstacles used during testing ROSbot

Tests Results

Before processing with the different cases results, it should be mentioned that the inspection of the CPU usage showed that only a small portion of it was in use while the majority of CPU was free. For the purposes of the competition, there was no need for RVIZ, so we excluded it from the launch file. Moreover, the ROS graph, which is shown in figure2, showed us that the connections between the topics and nodes were correctly configured.

In the first two scenarios robot seems to work correctly regardless of the shape and the size of the objects. Although the detection of the object was made from a distance of 0.9 meters, the change of direction took place at a distance of 60cm. Also, the rotation of the robot was dependant on which sensor was closer to the obstacle and what was the difference between the detected sensors distances. As such in the first scenario because both sensors detect the obstacle at approximately the same distance, robot select to pass constantly from the right. In the second scenario robot at first choose to pass from the right because the left sensor detected the obstacle closer than the right sensor plus the threshold. However, when the robot passes the first object and detects the second one because the right sensor detects this obstacle in a shorter distance than the left, it decides to the pass the cylindrical obstacle from the left.

In scenario three, the robot achieved to pass the obstacle, however, the fluctuation in the direction revealed an important software drawback. While the robot was getting closer to the obstacle, both sensors detect the obstacle. Because the difference between the two sensors distances is not big enough to pass the threshold, the robot makes a small turn to the right. During this turn right sensors continue to detect the obstacle while the left sensor doesn't. This happens because the left sensor points towards the haul of the glass obstacle. As a result, at the next iteration (after 100ms), the robot select to pass from the left. As such, the robot started rotating to the left. After a while, robot rotated again to the right, because the right sensor now didn't detect the obstacle. This created a fluctuation at the direction of the robot. However, the robot successfully passed. This didn't happen in scenario four. Robot was trying constantly to find a way from the right or from the left to pass the U-shape obstacle. When it happened to be very close to the wall, it went backwards for some centimetres and again toward the walls repeating the same functionality. ROSbot wasn't designed to pass such obstacles.

The robot is capable to detect effectively obstacles that are on the ground and take actions, but not obstacles which are not placed on the ground and more precisely higher than the height of ROSbot distance sensors. Although the use of the four distance sensors is an easy task and can simplify the coding procedure, it is vulnerable as the four sensors angle is not enough big to recognize higher obstacles. The solution to such a problem is the use of Lidar, which could provide us with valuable information for objects which are some centimetres above the surface.

The robot was designed appropriately for reaching the finishing line. That means, in case the robot loses its direction, it could spot the finishing line again and direct again toward this spot. Passing obstacles is not always an easy task and could mean that the robot has to change direction multiple times. It could mean that the robot could possibly direct to +- 90^o angle regarding the initial position or even in the opposite direction. For such cases, we used "rpy" to correct the direction of the robot. For testing its effectiveness, after placing robot toward the finish line and left it to move for some centimetres, we changed its direction by rotating the robot, for 180^o angle. After this rotation robot autonomously rotated again till its front side face the finish line. It must be mentioned, that there were no obstacles at the surrounding area. If there was, the robot will first pass them and after would correct its direction.

Conclusion & Further Work

To sum up, ROSbot was adequately programmed in a simple and extendable way in order to pass efficiently the obstacles and finish the competition. It can navigate in many different environments with multiple obstacles without colliding. Multiple cases have been included in the code that reassures the effective handling of difficult situations. The robot can successfully recognize an object from a distance and make a smooth turn, while in occasions it is very close to the obstacle, sharper turns are made. An occasion where the robot has to go back or stop have been included in the code. The robot has also a navigation system which helps it to find the right direction to complete the race. The robot gained the second place at the competition. This shows its efficiency of moving between obstacles and reaching successfully the final line in a short time.

The perception of the coding part, the included cases and the configuration values for the velocities were made by all the team members. However, the final implementation of the coding part was mainly performed by me. Also, i was responsible for restoring the current orientation of the robot, which was made by utilizing the "rpy" topic. At this point, it should be mentioned that there were attempts to use the Lidar and gain a 360^o view of the surrounding environment. However, the complexity of utilizing the array list of the detected obstacles, made us give up.

ROSbot code worked well at the specially designed environment of the competition. However, further work must be made to complete more challenging competitions and navigate more efficiently in the environment. First of all, ROSbot should continue with the initial orientation after passing the obstacle. Currently, the robot will not fix its orientation if this is not higher/lower than $+-70^o$ degrees. Robot when detects an obstacle try to pass it from right or left. A small rotation could make the front sensors to lose contact with the obstacle. As a result, according to the code, the robot will try to fix its orientation without having passed the obstacle yet. In order to exclude this occasion, the code was configured with high free rotation. However, a timer that would leave enough time to pass the obstacle and after activate the return to initial orientation would solve this issue. Moreover, an additional coding part could be implemented for identifying the u-shape obstacle and escape from it. Subscribing to the /pose topic and checking the returning position values could reveal if the robot is in the same region for much hour. If it happens, then the robot could make a big backward movement with a rotation. This problem, as well as other problems, could have been prevented if Lidar was in use. Lidar could provide a 360^o representation of the surrounding environment and take more efficient actions as the current code provide us with information for only the forward and backward view of the robot and only for a specific height.

action_controller.cpp   code :

```
#include <ros/ros.h>
#include <std_msgs/Float32MultiArray.h>
#include <std_msgs/Int32MultiArray.h>
#include <geometry_msgs/PoseStamped.h>
#include <tf/transform_broadcaster.h>
#include <geometry_msgs/Twist.h>
#include <std_msgs/String.h>
#include <opencv2/opencv.hpp>
#include <sensor_msgs/Range.h>
#include <std_msgs/UInt8.h>
#include <std_srvs/Empty.h>
#include <ros/time.h>
#include <geometry_msgs/Pose.h>
#include <geometry_msgs/Point.h>

int i = 0;
int timer = 0;
int counter=0;
float difference =0;
ros::Publisher action_pub;
geometry_msgs::Twist set_vel;
float init_dir=0;
float distFL = 0;
float distFR = 0;
float distRL = 0;
float distRR = 0;
float direction =0;
int first_time=0;
bool print_b;

tf::Transform transform;
tf::Quaternion q;
float deg2rad = 3.14159/180;


void imu_callback(const geometry_msgs::Vector3 &imu)
{
   static tf::TransformBroadcaster br;
   q = tf::createQuaternionFromRPY(imu.x * deg2rad, imu.y * deg2rad, imu.z * deg2rad);
   if(first_time==0){
      init_dir=imu.z;
      first_time=1;
   }
   direction= imu.z ;
   //std::cout << imu.z << std::endl ;
   transform.setOrigin(tf::Vector3(0.0, 0.0, 0.0));
   transform.setRotation(q);
   br.sendTransform(tf::StampedTransform(transform, ros::Time::now(), "base_link", "orientation"));
}

void distFL_callback(const sensor_msgs::Range &range) {
            counter++;
            if (counter ==10) {
                  timer = timer + 1;
                  std::cout <<"time = " <<timer<< std::endl;
                  counter = 0;
            }
            distFL = range.range;
            if(distFL >= 0.6 && distFR >= 0.6 ){
                       difference = direction-init_dir;
                       if( direction > init_dir +70 ){
                                     std::cout << " reorientating 1"<< std::endl;
                                     set_vel.linear.x = 0.25;
                                     set_vel.angular.z = -1.5;
                      }else if(  direction < init_dir -70 ) {
                                     std::cout << " reorientating 2"<< std::endl;
                                     set_vel.linear.x  = 0.25;
                                     set_vel.angular.z = 1.5;
                     }else {
                                    set_vel.linear.x = 0.35;
                                    set_vel.angular.z = 0;
                     }
                    //std::cout << " RPY initial " << init_dir  << std::endl;
                   //std::cout << " RPY current " << direction << std::endl;
                   //std::cout <<"straight " << std::endl;
            }
            if(distFL <= 0.2 || distFR <= 0.2   ){
                   if(distFL <= 0.2 && distFR <= 0.2){
                                  if(distRL <= 0.2 || distRR <= 0.2   ){
                                         set_vel.linear.x  = 0;
                                         set_vel.angular.z = 0;
                                        std::cout <<"stop going back" << std::endl;
```

```
                                }else{
                                        std::cout <<"Go back" << std::endl;
                                       set_vel.linear.x = -0.2;
                                       set_vel.angular.z = 0;
                                }
                        }else if (distFL<distFR){
                                set_vel.linear.x = 0.05;
                                set_vel.angular.z = -1.5;
                        }else if (distFL>distFR){
                                set_vel.linear.x = 0.05;
                                set_vel.angular.z = 1.5;
                        }
                }
                if (distFL <= 0.6 && distFL > 0.2 && distFR <= 0.6 && distFR >0.2){
                        if( distFR > distFL && (distFR - distFL) >= 0.2  ){
                                set_vel.linear.x = 0.3;
                                set_vel.angular.z = -1.0;
                                std::cout <<"big right " << std::endl;
                        }else if( distFL > distFR && (distFL - distFR) >= 0.2) {
                                set_vel.linear.x = 0.3;
                                set_vel.angular.z = 1.0;
                                std::cout <<"big left " << std::endl;
                        }else {
                                set_vel.linear.x = 0.01;
                                set_vel.angular.z = 1.5;
                               std::cout <<"Skid left" <<std::endl;
                        }
                }
               if (distFL <= 0.6 && distFL > 0.2 &&  distFR > 0.8   ){
                        set_vel.linear.x = 0.3;
                        set_vel.angular.z = -1.0;
               }
              if (distFR <= 0.6 && distFR > 0.2 &&  distFL > 0.8    ){
                        set_vel.linear.x  = 0.3;
                        set_vel.angular.z = 1.0;
              }
  action_pub.publish(set_vel);
}
void distFR_callback(const sensor_msgs::Range &range){
              distFR = range.range;
}
void distRR_callback(const sensor_msgs::Range &range){
```

```
            distRR = range.range;
}
void distRL_callback(const sensor_msgs::Range &range){
            distRL = range.range;
}
int main(int argc, char **argv)
{
  ros::Time::init();
  ros::init(argc, argv, "action_controller");
  ros::NodeHandle n("~");
  ros::Rate loop_rate(100);

  ros::Subscriber distFL_sub   = n.subscribe  ("/range/fl", 1, distFL_callback);
  ros::Subscriber distFR_sub   = n.subscribe  ("/range/fr", 1, distFR_callback);
  ros::Subscriber distRL_sub   = n.subscribe  ("/range/rl", 1, distRL_callback);
  ros::Subscriber distRR_sub   = n.subscribe  ("/range/rr", 1, distRR_callback);
  ros::Subscriber pose_sub = n.subscribe("/rpy", 1, imu_callback);
  std_srvs::Empty srv;
  action_pub = n.advertise<geometry_msgs::Twist>("/cmd_vel", 1);
  set_vel.linear.x = 0;
  set_vel.linear.y = 0;
  set_vel.linear.z = 0;
  set_vel.angular.x = 0;
  set_vel.angular.y = 0;
  set_vel.angular.z = 0;

  while (ros::ok())
  {
    ros::spinOnce();
    loop_rate.sleep();
  }
}
```

Distance.cpp code :

```
#include <ros/ros.h>
#include <geometry_msgs/PoseStamped.h>
#include <std_msgs/String.h>

ros::Publisher distance_pub;

bool firstIt = true;
double previousX, previousY;
double currentX, currentY;
double x, y;
double squaredVals, sqrtVal, distanceCalculated, finalDistance;
std_msgs::String msg;


void calculateDistance()
{
	x = previousX - currentX;
	y = previousY - currentY;
	squaredVals = pow(x, 2) + pow(y, 2);
	sqrtVal = sqrt(squaredVals);
	finalDistance = finalDistance + sqrtVal;
	previousX = currentX;
	previousY = currentY;
	std::cout  << " distance: " << finalDistance << std::endl;
}

void distance_callback(const geometry_msgs::PoseStampedPtr &pose)
{
	if(firstIt)
	{
		previousX = (int)(pose->pose.position.x * 1000.0)/1000.0;
		previousY = (int)(pose->pose.position.y * 1000.0)/1000.0;
		firstIt = false;
	}
	currentX = (int)(pose->pose.position.x * 1000.0)/1000.0;
	currentY = (int)(pose->pose.position.y * 1000.0)/1000.0;
	calculateDistance();
}


int main(int argc, char **argv)
```

```
{
            ros::init(argc, argv, "distance");
            ros::NodeHandle n("~");
            ros::Subscriber distancePose_sub = n.subscribe("/pose", 1, distance_callback);
            ros::Publisher distance_pub = n.advertise<std_msgs::String>("dist", 1000);
            ros::Rate loop_rate(500);
            while (ros::ok()){
                    distance_pub.publish(msg);
                    ros::spinOnce();
                    loop_rate.sleep();
            }
}
```