COMPSYS 726: PROJECT 2

PRACTICAL PROJECT
Obstacle Avoidance in ROS

Harvey Merton

Faculty of Engineering
The University of Auckland

3 May 2021

Table of Contents

List of Figures

1. Introduction

Mobile robotics are becoming exponentially more important in our modern world. From increasing flexibility and productivity in factories, to common household assistants such as the Roomba, mobile robots are making our lives easier in many ways.

An important function of mobile robots is obstacle avoidance i.e. how robots detect and move around obstacles in their path. Without this functionality, mobile robots cannot successfully navigate from one location to another without a high chance of damaging equipment or causing harm to people. The implementation of obstacle avoidance and other navigation algorithms can be broadly split into to paradigms: deliberative functionality (where a robot maps its environment prior to undertaking intelligent decisions) and reactive functionality (where a robot simply responds to its immediate external stimuli). In this project, a mostly reactive approach is employed.

One common framework used to develop robotic solutions is the Robot Operating System (ROS). This framework functions through a set of nodes that communicate via messages. In this project, the python programming language is used to develop the nodes which control a Turtlebot moving through an environment simulated on Gazebo. This report discusses an adaptation of the Bug2 obstacle avoidance algorithm developed to allow this functionality. Discussion begins with a general overview of the solution (both the ROS framework and the reasons for Bug2) before moving through more specific details of the implementation and the major challenge of using the LiDAR to track around an obstacle.

2. Solution Overview

At a higher level, the task of obstacle avoidance could be approached through either deliberative or reactive methods. Initially, deliberative methods were explored using inbuilt ROS mapping functions such as gmapping, and external libraries such as 'voxel grid'. The advantage of deliberative functionality is that it is able to deal with more edge cases easily, and it allows more optimised path planning than reactive methods. Because a map of previously encountered obstacles and obstacles in the distance is kept, the robot is more easily able to avoid getting itself into difficult situations before they occur. However, deliberative functionality requires much more processing power for map building and path planning than its reactive counterpart. Further, as gmapping proved to be difficult to interpret, and the easy occupancy maps generated by the 'voxel grid' package required additional installation (restricted for this project), the simpler reactive approach was selected.

2.1 Nodes

To make the solution more modular, reusable and extensible, it was distributed across a number of nodes (seen in Figure 1). This enabled processing of data to be done before it reached the main obstacle avoidance node (in the case of the sensors) and actions to be performed without lower-level control from this node (in the case of the TakePhoto node).

Figure 1 shows the custom nodes in boxes and the messages that commu-

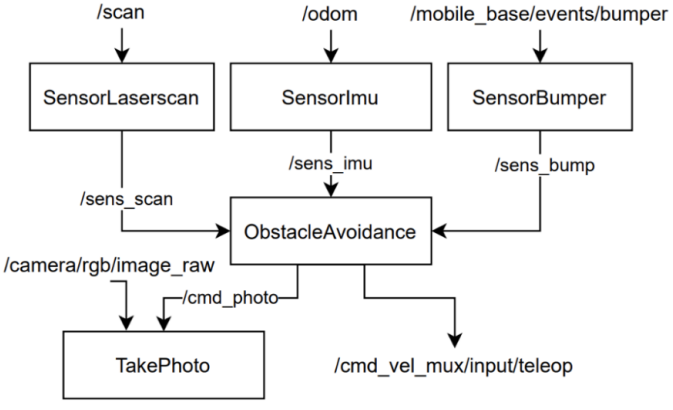

Figure 1 ROS Nodes and Messages

nicate between them on the arrows. The messages on the ends of the arrows are subscribed to/published by nodes running in background of ROS. Initially, a movement node existed to separate the obstacle avoidance (OA) node from direct interaction with the teleop message. A custom message was developed to allow both speeds and exact angles/distances to be sent to this node, to which the movement node was designed to react accordingly. However, it was found that because this node was implemented using a topic (i.e. publish/subscribe model) rather than a service, and some loops were present in it, there was a lot of lag between commands sent by the OA node and their action in the movement node. As it was easier to implement the required functionality in the OA node, and not all of the functionality attempting to be developed was actually required (e.g. turning by an exact arbitrary angle), this node was put aside for future development.

A brief overview of the auxiliary nodes is given below:

- **SensorLaserscan** - Takes in range data from the 2D LiDAR (modified depth camera) in the form of a list and outputs a list of 0's and 1's across the LiDAR's field of view (FOV). The output list contains 1's where a gap large enough for the robot to fit through exists and 0's where an obstacle or small gap is present (see Section 4).
- **SensorImu** - Subscribes to position and angle data provided by the odom topic which is all relative to the robot's starting position and pose. As the odom topic outputs angle data in quaternion coordinate system, this node also converts this data into a Eulerian coordinate system. In this system, the robot's yaw angle is expressed in degrees measured counterclockwise from the x axis (connects the robot's starting position to the end wall with the door). This way, the robot always knows where it is relative to the centreline, and which direction it must travel to reach the end of the course.
- **SensorBumper** - Simply subscribes to the bumper event and publishes the same information to the OA node. This node is purely a placeholder to allow easy implementation of any bumper information pre-processing desired to be implemented in the future.
- **TakePhoto** - Captures and saves images when commanded by the OA node. Takes care of image counting, naming and saving in the correct directory.

Although the course could likely be completed without the aid of the LiDAR, use of the LiDAR means that gap sizes and obstacles further away can be detected. It also allows some intelligent decision behind which direction the robot should turn upon encountering an obstacle (robot turns in the direction where less of its FOV sees an obstacle). These enhancements make the robot's response slightly better which can make a big difference in some situations (e.g. turning the wrong way leads to a big detour). Because the LiDAR is set at a specific height and has a dead zone of 0 - 0.3 m, the bumper still has to be implemented to allow the robot to react to all obstacles (some obstacles e.g. the short bricks, fall below the LiDAR's FOV).

2.2 Messages

To keep the solution as simple as possible, all node interactions were implemented using topics (i.e. no services), and no custom messages were used. This reduced points of failure in the solution and reduced development time. However, as the TakePhoto node took some time to respond, flags had to be implemented to ensure that only one photo of each obstacle is taken. Had a custom service instead been used for this interaction, the use of flags could have been avoided, potentially increasing solution elegance.

3. Obstacle Avoidance Algorithm

The Bug2 obstacle avoidance algorithm assumes that the destination of the robot, as well as the robot's current position, is known at all times. The robot then attempts to follow an imaginary direct line between the start and finish positions, following the side of any obstacle it encounters along the way until it is back on this line. In this project, as the exact xy coordinates of the door are not known prior to execution, the algorithm is adapted to cause the robot to continue moving along the centreline while it is lined up with it. This original implementation is inspired by [1].

At the highest level, the obstacle avoidance algorithm is implemented with a simple two-state Moore-type finite state machine (FSM, see Figure 2). The robot starts in state one (following the centreline) until it hits an obstacle and transitions to state 2. In state 2, the robot follows the edge of the obstacle until it comes back on the centreline and so transitions back to state 1. The robot will continue to follow the obstacle if it is not ahead of the point of initial impact or if it turned around and is coming back along the obstacle. The second case is to prevent the robot from making the same incorrect direction decision twice after abandoning following in the other direction.

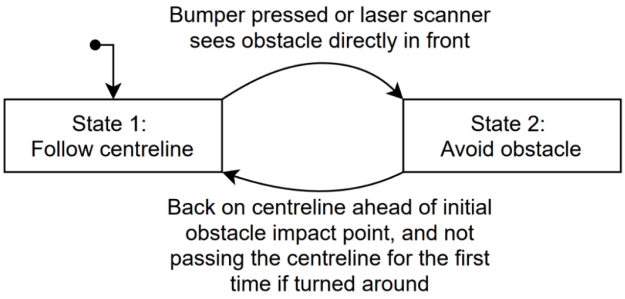

Figure 2 Obstacle Avoidance Finite State Machine

Key Assumptions:

- **Obstacle isn't stuck on an object for too long**. The /odom message tracks the movement of the wheels rather than the exact xy position in a map. Should wheel slip occur due to getting stuck on an obstacle, the odometry reading will be incorrect.
- **No object except the final wall is longer than 2.5 squares back in the x direction**. If the robot is following an obstacle and reaches 2.5 squares behind its initial obstacle hit point, it will turn around and follow the obstacle the other way. Should an obstacle be longer than this, the robot may abandon the correct following direction before it has completely overcome the obstacle.
- **Obstacle following arc won't result in the robot getting caught on obstacles that don't cross the centre line**. When the robot is following the obstacle with the LiDAR and it loses sight of it, the robot will continue to arc in the approximate direction of the obstacle (to enable smooth following). If another obstacle is close by, the robot could run into that obstacle with its bumper and start following it instead. This could potentially lead the robot away from the centreline thus resulting in never-ending following of an obstacle (i.e. doesn't leave state 2).

Flowcharts for the logic in state 1 and state 2 are shown in Figures 3 and 4 respectively. The 'follow centreline' logic is reasonably straightforward; the robot always acts to keep itself on the centreline and pointing towards the exit wall. The only potentially abstract part is the driving in arc logic. If the angular deviation from the line becomes too great, the robot must stop and drive perpendicular back towards the centreline before continuing. This takes longer than driving in an arc but the combination of both actions means that the robot maintains a reasonably straight trajectory while getting to the door as fast as possible.

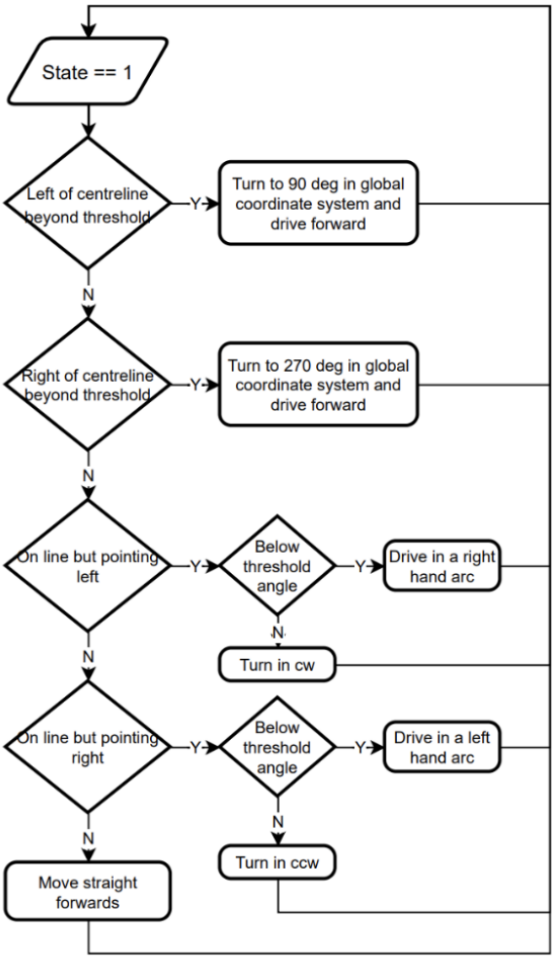

Figure 3 Follow Centreline Logic Flowchart

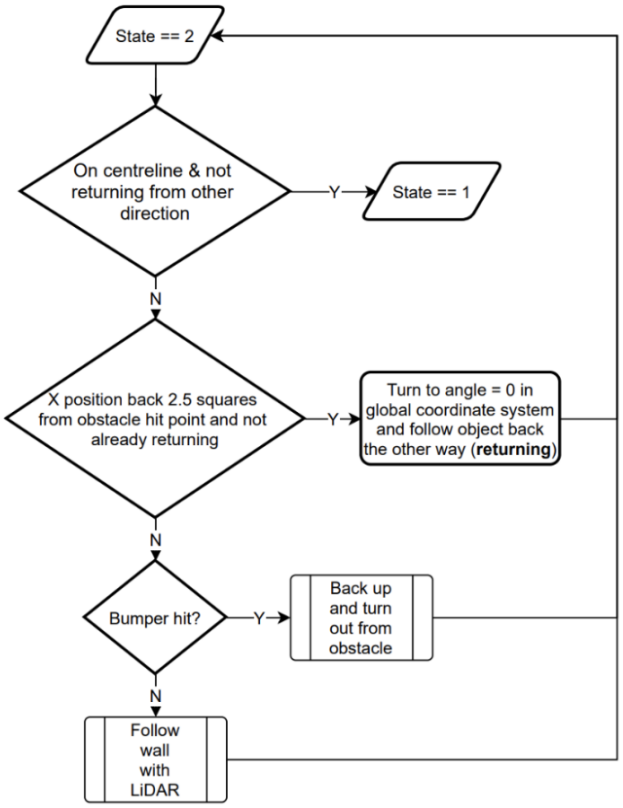

Figure 4 Avoid Obstacle Logic Flowchart

The 'obstacle avoid' logic is slightly more convoluted. The key part in this algorithm is how the robot turns around if it follows an obstacle to a point 2.5 m behind where it first impacted that obstacle. This logic makes the robot more robust to an initially bad decision on which side of the obstacle to follow (one may lead to a dead end while the other is OK). It also helps the robot get out the door at the end of the world, otherwise it would follow the wall back down to the start. The other parts to notice are the sub-processes to avoid the wall with bumper and with LiDAR. The bumper implementation is simple: when a bumper is hit, back up, turn out slightly and continue. Conversely, following the side of an obstacle with the LiDAR was the most challenging part of the project and so is discussed in Section 4.

4. Challenge Discussion: Following Obstacle with LiDAR

Rather than simply using the bumper to navigate around an obstacle, the LiDAR was also used to allow a smoother obstacle-following response. This was challenging for a few reasons. Firstly, the LiDAR's FOV only extends 30 deg either side of the Turtlebot's direct front. As the LiDAR also has a dead zone up to 0.3 m, this makes it difficult for the LiDAR to see the obstacle when it is closely following it. Secondly, partially due to the first reason, the robot only has a fine range of linear and angular speed ratios at which it can closely follow an object. As all objects vary in shape and size, it is difficult to tune following speeds to fit all objects. Finally, if multiple objects are consecutively lined up, it can appear that a gap is present when it is not large enough for the Turtlebot to pass through. This often resulted in the Turtlebot getting stuck.

To resolve the problem, each of the sub-issues described above were approached independently. Firstly, to simplify development, rather than using the raw ranges output from the laser scanner, these values were thresholded in the SensorLaserscan node (see Section 2) i.e. if an obstacle is closer than 1 m in a sample point, that point becomes 0 (for obstacle), otherwise it is a 1 (for gap). By looking at the number of consecutive ones in the thresholded list, an angle between the robot and the two obstacle edges (between a and b in Figure 5) can be estimated. Finding the average distance to the centre of the gap and using the cosine rule, an angle required to fit the robot through can be calculated. If the actual angle is greater than the required angle, the gap is legitimately large enough for the robot to pass through and so the 1's are left in the list. If this is not the case, the 1's are replaced by 0's leaving the robot thinking that the small gap is simply a wall. This step was inspired by the vector field histogram approach but rather than using statistical representations, a simple yes/no threshold was used. This was found to be very successful at removing almost all small gaps while keeping ones large enough.

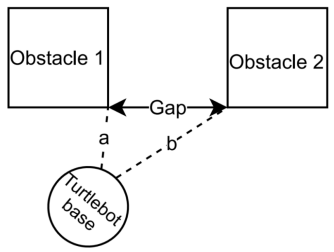

Figure 5 Laser Scanner Gap Size

Once the LiDAR's FOV was thresholded, the robot's response had to be determined to keep it close to the wall. Due to the small FOV, it is inevitable that the LiDAR eventually loses sight of the wall as the robot gets either too close or too far away. It was determined that it is better that the robot get too close, as it can respond with the bumper, than too far away when it would lose the obstacle. For this reason, the responses are chosen to always move the robot back towards the wall and favor collision over missing the obstacle. Defining a controller that modulated the angular speed proved to be very challenging to tune and always result in the robot hitting the obstacle, or losing sight of it, too regularly. To overcome this, the LiDAR's FOV was split into five regions (see Figure 6, inspired by [2]) which have discreet defined angular and linear rates when an obstacle is seen within them. Eventually, through trial and error on obstacles of many different sizes, speeds that worked well enough for a range of obstacles were found. Although this works, it has proven to be very difficult to change the speed at which the robot moves around the obstacle. For this reason, the robot is quite slow when using the LiDAR to track around some objects - more tuning is to be done here.

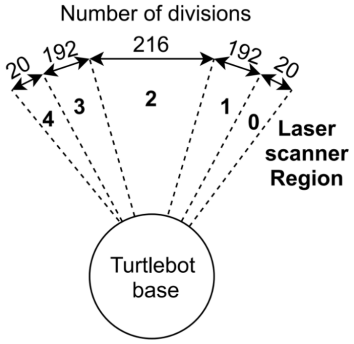

Figure 6 Laser Scanner Regions

5. Conclusion

COMPSYS726 project 2 required students to implement an obstacle avoidance algorithm using Python, ROS and Gazebo. To facilitate modularity, multiple nodes were used to achieve this: laser scanner, IMU, bumper, photo taker and the obstacle avoidance node. Although OA could have been achieved more robustly using mapping and deliberative functionality, this project used a mostly reactive approach by implementing a modified form of Bug2, and a gap detection algorithm inspired by vector field histogram. The most challenging part of this approach was using the LiDAR to follow an obstacle due to limitations on the LiDAR's FOV and the difficulty of determining gap size which was solved using the cosine rule. Overall, although this project was very challenging and time-consuming, it has been extremely rewarding and a great learning experience.

References

[1] M. Arruda, "Exploring ROS with a 2 wheeled robot #12 bug 2," Jul. 2019. [Online]. Available: https://www.theconstructsim.com/exploring-ros-with-a-2-wheeled-robot-12-bug-2/

[2] ——, "Exploring ros using a 2 wheeled robot: Obstacle avoidance," May 2019. [Online]. Available: https://www.theconstructsim.com/exploring-ros-2-wheeled-robot-part-5/