Integrated Group Robotics Project

Group: Shayne Shaw and Jack Rome

Reactive Motion Planning – Adapted Dynamic Window Approach

**Abstract**

The turtlebot uses a modified dynamic window model to evaluate the best path to the goal. Goal coordinates are set as a global variable and the robot will plan and travel in sub-goals based on 0.5m increments until it’s within an acceptable range to the goal.

While it was found that the robot was able to successfully navigate to a specified coordinate without collision, issues with the odometry subscriber’s angle caused the robot to spin in circles as it had no angle to orient itself. Another issue comes from a case where there is an obstacle taking up the robot’s entire range of view. The result of this is that the robot decides that straight forward is the best option to goal as every scan reveals an imminent collision and this path is the closest distance to the goal heuristically.

**Introduction**

The dynamic window approach uses…..

Subscribers:

(“/odom”,Odometry, newOdom)

The odometry subscriber is used to collect the x and y position of the robot in metres using the command ‘msg.pose.pose.position’. The odometry also returns the rotation of the turtlebot, however, this is initially returned as a quaternion and so a function from the ‘tf’ library is used to convert the quaternion value to a Euler value so that the rotation on its z axis can be obtained (in radians). This is then converted into degrees as both of us find that easier to work with.

(“/scan”, LaserScan, steering)

The scan subscriber is used to take in scans from the hokuyo laser scanner mounted on the front of the turtlebot. The scan returns an array with the distance to the closest obstacle to the scanner (in metres) within a 180-degree range. It was found the Gazebo simulator uses a scan with around 720 values whereas testing the code on the turtlebot revealed a scan with 512 values. The array lengths for the data ranges in this subscriber’s call-back needs to be adjusted when switching between these two testing methods.

Publishers:

('/cmd\_vel\_mux/input/teleop', Twist, queue\_size =1)

The teleop publisher is used to give the movement signals to the turtlebot. This is achieved using the Twist() import from the geometry\_msgs.msg library. A global variable ‘speed’ is used to store the Twist() instructions and this is published at the end of the program, just before the spin.

('/mobile\_base/commands/reset\_odometry', Empty, queue\_size=10)

The ‘reset\_odometry’ publisher is used on start-up to ensure that all of the odometry values are localised to zero at the beginning of the program. This publisher is called within a while loop with a timer of 1.5 seconds. It was found that having long lengths of time to reset the odometry gave initial values closer and closer to zero. 1.5 seconds gives the best balance between start-up time and practical local values with the x and y variables typically being in the micro metres.

**The Path Planning Model**

The program uses the input from the hokuyo laser and splits the scan into seven regions. For each region, a new set of coordinates is set to 0.5m in that direction. The minimal scan value in each region is compared and the distance to the goal is also compared. If the minimal scan value is less than 0.7m then the value of the distance to the goal, for that region, is multiplied 20 so that it is not considered as a valid option. The comparison distance is set to 0.7m rather than 0.5m to give the robot some space and to account for braking speed, refresh rates and other factors that could cause error A new sub goal is then set to travel in the resultant best direction. Once this sub goal is achieved, this model repeats for a new sub goal until the final goal is reached.

Turtlebot

Scan Radius 0.7m

Path planning Radius 0.5m

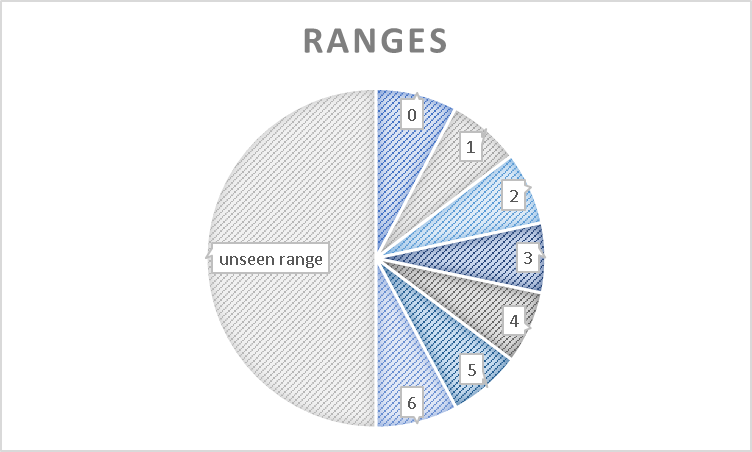
0O

-90O

90O

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Index | Data Range | Number of values | Angle of arc (degrees) | Angle of path |
| 0 | 719:608 | 111 | 28 | 65 |
| 1 | 608:509 | 99 | 25 | 40 |
| 2 | 508:409 | 99 | 25 | 15 |
| 3 | 408:313 | 95 | 24 | 0 |
| 4 | 312:213 | 99 | 25 | -15 |
| 5 | 212:113 | 99 | 25 | -40 |
| 6 | 120:0 | 112 | 28 | -65 |

Each predicted path can, therefore, be defined as:

Pi(x, y) =

where x and y represent the robot’s current position and i represents a region from 0-6. The angle Θ will therefore change based on the region and so seven unique paths are calculated to be tested. The distance to the goal can be expressed as:

Where is the lowest scan value returned within that region (in metres). This distance is compared and the smallest path is selected to be the sub goal.

Comments

* The program works smoother when using a lower ros rate (e.g. 5-10Hz). This is because the turtlebot has more time to turn before the call-backs are updated.
* It was found that the error in the difference between the robot’s current angle and the angle required to turn to the goal, is best set as ±4.5 degrees. This is the best compromise between speed and accuracy in the programs ability to turn in the direction of a goal.

The angle from the odometry will sometimes not be published and so the turtlebot will rotate on its axis forever. Gazebo simulator will need to be shutdown and restarted to fix this issue.

Appendix

Odometry callback (youtube video inc.) - <http://www.theconstructsim.com/ros-qa-know-pose-robot-python/>

Pseudo-code used - <https://adrianboeing.blogspot.co.uk/2012/05/dynamic-window-algorithm-motion.html>

Odometry reset - <https://answers.ros.org/question/203088/reset-turtlebot-odometry-in-a-python-script/>