Robot Code Documentation

The code for this project involves using ROS in order to communicate with other nodes in order to control the simulated turtlebot in Gazebo. This is accomplished by publishing topics and subscribing to topics being published by other nodes. In this document, we will examine the reactive architecture used and how the code is structured and how it actually works.

The program is based on the layered control system described in Brook's paper. That is to say, behaviors that have a higher role in the layer can subsume lower levels whenever necessary. In the program, all the modules are implemented as separate functions that will be running simultaneously and this architecture allows modules with a higher priority to subsume lesser prioritized modules to avoid interference whenever a higher priority module takes control in a certain situation e.g. avoiding an obstacle while suppressing the drive and random turn functionalities to allow the avoidance behavior to be executed. These different modules implemented as functions behave as layers, which, as mentioned before, have different priorities. These priorities, however, are mentioned in Brooks' paper as "levels of competence." In our implementation, the zeroth level of competence is handled with the drive and turn functions in order to produce our "wander" module. The other levels are as follows going in ascending levels: obstacle avoidance (using the avoid and escape functions), keyboard, and halt. For example, we have the keyboard module, which will use keyboard teleop to determine what keystrokes drive the turtlebot and it takes over the wander module and even the obstacle avoidance module so that the user can have total control without the interference of these modules. In this way, the control system is formed. For these reasons this architecture was chosen.

Next, the actual structure of the program will be outlined and is largely based on the reactive architecture that we have chosen. The program was written using object oriented c++ and each behavior is programmed in its own function belonging to the Explorer class. As a result, although the main function has no direct access to these functions, they can be easily ran by the main function by calling explore() after creating an instance of an Explorer object. This creates subscriptions to topics being published by other ROS nodes allowing for callbacks and runs other independent functions in their own threads allowing each function to be active at the same time. While these functions are running at the same time, it is possible for functions with a higher priority to suppress some of the lesser functions whenever an event occurs that requires the higher priority functions to take action. Because each behavior is designed as its own function and they are all running simultaneously, there needs to be some communication between these behavioral modules such that the priority levels are adhered. That is, one module should be able to send a suppression message to other lesser functions so that they can let the higher priority function do its job without interference. For example, the robot must stop driving before turning – once the distance of one meter is reached, driving is suppressed by the turn function being activated. For this Booleans, defaulted as true, are used to determine which functions can run and can be switched on/off to enable/disable functionality In addition, information such as distance travelled is stored in order to activate higher priority functionality whenever a condition is met i.e. turning after one meter traveled. It is in this way that modules can communicate and suppress each other when the need arises. Using this approach, we can implement Brook's subsumption architecture.

Now that the code structure has been outlined, some of the implementations can be explained. We have the turn and drive functions which are the only two functions that run in separate threads since they are the only functional functions that do not subscribe to anything. The drive function only published commands for the turlebot to move forward while updating the distance travelled; turn selects a random direction in which to turn 15^o after one meter has been travelled and disables driving while doing so. Next, we have the other functions, avoid, escape, keyboard and halt, which subscribe to topics and are called whenever they receive a message. Halt is the simplest since it terminates the program whenever the turtlebot has bumped into something and has the highest priority i.e. it is never disabled. In the middle, we have the obstacle functions avoid and escape. The functions receive pointcloud messages from simulation and react whenever an obstacle is detected. This is done by getting the average distances of the object in question from the left and right sides. If the object is symmetric, i.e. right and left distances are about the same, then escape is activated and rotates the turtle bot 200^o otherwise it will activate avoid and keep turning away from the obstacle until it no longer is detected. Boolean variables are used to determine whether the turtlebot is avoid/escaping in order to ensure that multiple avoidance maneuvers aren't being run simultaneously. Finally, we have keyboard, which listens to the teleop keyboard messages. Whenever a message from the teleop keyboard is received, all functionality, except halt, is disabled while move is being published. Even though turtlebot can still move whenever teleop keyboard is running and user has not pressed key, the movement will be buggy since teleop keyboard publish nonmoving commands constantly. As mentioned before, all of these functions are running at the same time and are the heart of the program that keeps turtlebot exploring through the unknown world.

The program implements the Brook's subsumption architecture using c++ and ros to allow turtlebot to move and react to its environment. This is done by getting sensor data from topics being published and running all the functions simultaneously using threads and subscribers.