**Robot Code Documentation**

The code that navigates the robot throughout the world is meant to be reactive in nature. That is, it does not choose where to go, but rather, it moves and turns (in our case, it will turn in some arbitrary angle after moving one meter). The reactive part of the code base is that when the robot senses an obstacle with its visual component (via pointcloud from the Kinect), it will respond by turning away from it. This is the only hand which deliberatively forces the robot in some direction. Despite that description, the overall mechanism is solely reactive in nature.

The way the code is built is that each behavior is programmed in its own function. As a result, they can be easily called by the main function when the robot is required to fulfill a certain task. At the same time, it is possible to stop any one behavior in case a behavior with a higher priority, such as keyboard input when the robot is wandering around, to take over. This functionality is similar to the layered control system described in Brooks’ paper. That is to say, behaviors that have a higher role in the layer can subsume lower levels whenever necessary.

Because each behavior is designed as its own function, and subscribers are designed to listen to them individually, there may need 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 another, and that subscriber will be able to hear it and take the appropriate course of action henceforth. For example, the robot must stop before turning – in the drive function of the code, once the distance of one meter is reached, driving is suppressed and the turn function activates once it is published. It is in this way that modules can communicate and suppress each other when the need arises.

This architecture follows Brooks’ subsumption architecture. These different modules and functions behave as layers, which, as mentioned before, have different priorities. These priorities, however, are mentioned in Brooks’ paper as “levels of competence.” In this code, the zeroth level of competence is handled with the drive and turn functions in order to produce our “wander” module. The level with more precedence is the keyboard function, which uses teleop to determine what keystrokes drive the turtlebot. Because it has a higher precedence, it takes over the wander module. In this way, the control system is formed. It is notable that the function with the highest precedence is the halt feature, which forces the turtlebot to shutdown upon contact with any obstacle.

The code itself utilizes several private class variables that function as a sort of “switch” for each module. These Boolean values determine whether a function should be running or if it should be repressed. For example, the keyboard function turns off many of these switches (in effect, repressing the other modules) by setting these values to false so that the teleop commands can be read. It also ensures that these values return to their true state when the keyboard is no longer in use. The turn function also utilizes this concept as it has to suppress the drive function. In a sense, when talking about the subsumption architecture in reference to these functions instead of modules as a whole, the drive function would be level zero.

The subsumption architecture is most visible when dealing with the escape and avoid features of the code which have their own classes. While they are not the highest priority, they are both concurrently running with the wander module when watching for obstacles – and on detection, they proceed to take control of the robot. The code uses the point cloud to check if anything is in the way of the robot’s path, and then updates the status. When this status is updated, the functions check to see if either of their conditions are true, and, when true, the function turns off (that is, sets to false) the class Booleans that allow the robot to drive and turn (and, to ensure that they do not conflict with each other, the escape function sets off the avoid variable). From there, they both make the robot rotate (escape has the robot rotate more than the avoid function due to the requirements of the project).

In conclusion, the reactive architecture of the code base can draw much relation from the subsumption mechanism described in Brooks’ paper. Each function is generally in use in its own thread allowing for most levels to work concurrently. These levels, however, can have priority over the other levels and suppress them in order to take full control of the robot whenever the conditions are ripe for progress.