Robot Code Documentation

Style Guide Conformance:

This is a simple module with a single node, *project2*, and as such our file is called *project2.py*.

Since our last project, we have reorganized the project to have a more Object-Oriented structure. Each subscriber handler has been split into its own module, containing a single class with a *handle* method. These files are placed in the top level of the src/ folder, as they need to be exported for other nodes to use. The working copy of these files is, however, in src/scripts/. Moving the files is accomplished by running the *files.sh* script once updates to the code have been finished
(Sorry if this clobbers someone else's project name and causes any issues)

Architecture Overview:

We have implemented a form of reactive architecture that uses a subsumption system. The node keeps track of a priority level, where each behavior is allowed to operate only at specific priorities. Somewhat counter-intuitively, a higher priority level indicates a less important behavior. Behaviors may be blocked by the execution of another behavior that has a lower priority level. It is usually the job of the lower level priorities to indicate when higher level priorities may resume.

Through this system we have ensured that only 1 priority will run at a time – within 1 tick of the ROS topics other priorities should stop executing. This allows, for example, the bumper handler to lower the priority level and immediately stop all other velocity commands. Given the reason for the bumper handler, this is quite important.

This subsumption is allowed to happen even while other levels are running, as each behavior listens on its own thread – but has access to the shared priority level. We can still receive bumper data, for instance, even while odometry messages may be being processed.

All behaviors except for the lowest priority (highest priority level) occur in reaction to a ROS topic, hence the reactive component of the architecture. Each subscriber is executed in its own thread and will run continuously along side the forward motion behavior. Whenever a message is queued for one of the subscribers, the callback for that subscriber can choose to make a change to the priority level in response.

The interactions between levels will become clear in the following section.

Code Overview:

We begin by importing the various modules needed for our code to execute. Importing the common module then creates the *project2* node, with the anonymous option set to allow ROS to choose names for our listeners itself. We set the rate to 10 hertz.

The common module also creates a publisher on the '/mobile_base/commands/velocity' topic. This publisher is re-used by most aspects of the node. Three global variables are then created that are shared by all parts of the project. These three variables store information on the current priority (discussed previously).
2

uniformTurn (common module)
We next implement a method to allow for in-place turns, using a uniform distribution of possible angles. This code was developed through the use of internet resources (a citation is available in the code). This method computes relative angles and speed necessary to turn the robot in-place. This method was placed into the common module to allow its reuse across multiple modules.

BumpHandler
This module was also developed from online resources (a citation is similarly available in code). This particular implementation has been modified to set the priority level system we have implemented to the zero value, stopping all other behaviors.

TeleHandler
This module is triggered when tele-op commands are sent to the robot. It doesn't do any handling of the commands itself, however. This module changes the node priority level to 1 so that other commands don't interfere with the teleop while it executes. The priority level is reset to 5 after the teleop is complete. This has a nice side effect of allowing the user to escape from a collision induced halt by providing teleop commands to escape. Do note that this will not allow you to continuously re-ram the wall, as the bot will still be stopped by bumper events.

ObstacleHandler
This module handles obstacle recognition and determines if the obstacle is symmetric. It then calls additional class-level methods that properly handle the priority level and develop a strategy for avoiding or escaping the obstacle.

The recognition system is implemented by receiving a set of laser scan data from the robot. This laser scan contains ranging information as an array of 640 ranges, each range being ~0.1 degrees from the previous. The array represents data from right to left. If any point is within 0.6 meters (the requested range + some space to allow the robot to stop / maneuver), additional processing is performed. As this data can be easily represented as a set of left-right pairs, we iterate down the later half of the array. We compare each L-R pair and ensure that the data is within the minimum and maximum ranges (i.e., the data is not NaN). This last step ensures that we don't register readings for objects that are far enough away to not be an issue at the moment. The *isnan* check proved a bit intensive during our testing, and so we've

opted to use Numpy's alternative. (Numpy should have been installed as one of ROS's dependencies)

We then calculate a symmetry score by subtracting the sum of the left values from the sum of the right values. If this score is less than 30 (value determined by simulated testing), we assume the object is symmetric and attempt an escape. Otherwise we attempt to avoid the obstacle.

We will constantly receive continuous updates to this scan as part of the avoidance mechanism, so we only allow the algorithm to run if the priority level is not 2. This ensures that we can continuously avoid until we no longer need to. Once any criteria for continued processing of scan data become invalid, we return to priority level 5.

Avoid

If a non-symmetric obstacle is detected we stop and briefly execute a small, in-place turn. Due to our setting of the priority level here, this step will be repeated as needed until there is no longer an obstacle in front of the robot. Once it is clear, the priority level is raised and we are able to resume execution. Avoid is executed at priority level 3.

Escape

Similarly, we attempt to escape from obstacles by stopping and executing a turn within a range of 30 degrees around 180 degrees. Escape is executed at priority level 2.

OdometryHandler

In our odometry handler we compare our current pose to the previously received pose. Given the odometry estimates we receive, we are able to estimate our current distance traveled. After converting this distance into feet we are able to move into priority level 4 after every foot of movement. This stops the regular forward movement, and instead replaces it with a uniform turn. This allows random turns in a uniform distribution between -15 and 15 degrees.

ForwardsHandler

We finally have our lowest priority behavior. The forwards behavior will attempt to continuously move forward at a low speed, so long as ROS is running and we are in the highest priority level (5).

Listener Registration

We register subscribers to the ROS topics that our node requires to run. We have lowered the queue_size of the laser's subscriber to 1, so that we are always operating on the most recent laser scan available. This reduces the apparent latency of the scan significantly – and solved an issue with the scanner registering ghost objects during our early testing.

Exit Handler

This is registered as a signal handler for SIGINT, allowing the loop in our program to be terminated by user input. This is mostly useful for testing the script in isolation, as execution through *roslaunch* causes some issues here. The script should be exited through *rosnode kill* if that is the case.

Main

We register our signal handler and then begin a loop that will call the forwards behavior until it is either interrupted by user input or the priority level drops to the bumper level.