My first effort when I approached the open-loop stdr_control code was to rearrange its structure. I am familiar with general C++ syntax, but only to the extent that it overlaps with Java, MATLAB, Arduino, and other similar languages. I also thought it would be easier to manually contrive an open-loop sequence if the drive and turn tasks could be completed in one line. I didn't really understand how scope works in C++, with namespaces and whatnot, so I couldn't figure out how to write a simple 'drive' method at the end of the file, since it needs persistent access to the ROS Rate object, the publisher, and the twist command.

I don't want to have to reinitialize each of these objects every time the robot needs to alter its motion, so instead I created a class that could contain these objects as attributes. This class has a constructor which constructs the necessary ROS objects so they can be used by other public methods. It has a drive method that accepts as parameters a forward velocity, an angular velocity in yaw, and a duration. It commands the robot to move with these velocities for the duration given and then returns. There is also a halt method which commands zero velocity several times and then returns. After several attempts, I got this code to a point where it made sense to me and didn't throw any compile or runtime errors.

Unfortunately, when I ran it, the robot didn't do anything. I didn't want to waste any more time trying to diagnose the problem, so I saved it as a backup and started anew from the base code on Dr. Newman's github. I just copy/pasted and slightly altered the lines that tell the robot to move or turn for specific amounts of time. I quickly found that the inconsistencies due to timing are too significant to let the robot consistently avoid walls and find it's goal. I could have reduced the timestep and attempted to improve fidelity of the open loop control, but I decided to go a different route.

While we don't have any sensors for this project, we do know that the robot can't drive through walls. Using this knowledge, I increased the robot speed to 3m/s so it will almost always hit a wall, at which point it stops stop. In this way, I can minimize the accumulated error from a sequence of imprecise movements, because I can bet on the precise position where the

robot will stop. The only uncertainty now lies in the rotation, and the few times where I have the robot drive over short distances without a collision. With this method I was able to find the goal with some consistency, and this method reflects the attached video and final code on GitHub:

https://github.com/jdc183/Mobile_Robots_PS1

I understand that this method would be unacceptable for almost any real robot, as physical collisions can cause damage or injury. But at the same time, I would never program a real robot with open-loop control in the first place. The constraints for this problem are already artificial, so there's no reason to artificially impose realistic constraints.