Dean Fletcher

Report – Lab 6

Video: https://youtu.be/rir9d149cD4

Task 1: Wall Mapping

For this task our goal is to traverse maze until all cells are visited. For each newly visited cell scan the walls to map the environment. This task builds on the many procedures developed over this semester. I took the localization code and incorporated it with maze wall following logic. With the help a my PID functionality and localizing at every new cell we can scan for walls. Localization keeps up with the current (x, y), cell number, and the column/row that cell resides. If a new cell is found, then mark with an 'X' and if there are more cells to visit keep searching. Once all cells found then prompt the maze wall configuration.

PID:

Kp = 0.1

$error = desired - measured$

$control = Kp * error$

$saturation = f_{sat}(control)$

$$f_{sat}(control)\begin{cases} if\ control > max & control = max \\ if\ min \leq control \leq max & control = control \\ if\ contorl < min & control = min \end{cases}$$

Kinematics:

Known: Velocity, Distance

Omega (w) = v/R

Turn velocity = w(R + axle/2)

Wheel Velocity = velocity / wheel radius

Drive Time = distance / velocity

max velocity RPS: $-6.28 \leq max \leq 6.28$

X, Y += distance traveled and corrected with distance sensors if possible

Rotation:

Wheel Velocities: $\text{max} = (\pm)6.28$

Wheel Velocities: $(\text{max}, -\text{max})\ or\ (-max, \text{max})$

$\text{Time} = \frac{\text{Distance}}{\text{Velocity}}$

If IMU degrees equal target degrees:

- Return

Task 2: Path Planning with Wavefront Planner

Task 2 builds on task 1 though the difference is we have the wall configuration ahead of time. The goal here is to perform a 4-Point connectivity wavefront planner algorithm. What this does is send out a ‘wave’ from the goal to the starting pose. Each wave increases in cost to drive per cell. The logic is that once you have the grid setup you can simply follow the decreasing values until you reach your target. The way I managed to do this was to send waves and is similar to a Breadth-First-Search. First, I created a list of all valid neighbors. Then as I visited each neighbor a hash stores the lowest cost with the cell as a key. Once that was performed then I reverse sorted the map and built a path off the valid moves (ie the cell before must be a neighbor). To follow this path, I applied the wall follow logic from task one and similar cell movements as normal. Based on heading and the target cell the option is either up/down for a row or left/right for a column. Once the current cell is the goal, we are complete.

Kinematics were the same as the previous task. This task required more of an algorithmic approach.

Conclusion:

This was a fun, and challenging, lab. I found that combining previous code into one controller added off-by-one errors that were difficult to track down. Path planning took a lot of playing around with data structures to find one that performed the way I wanted it to. I found Python dictionaries to be the most useful. Though I’ve tested many edge cases and found many bugs along the way I feel there are still improvements to be made. For one I would really like to make the project more abstract by applying OOP concepts. This would really help clean up the code and make it more robust. Unfortunately, time did not permit but it was a good experience, nonetheless.