Igor Amidzic
Nicholas Boswell

Lab 2

Program Files

Main.py

Asks the user to select a task to run (calibrate speeds, test distance sensors, move forward until close to wall, follow parallel to walls, follow walls).

Encoders.py

Contains methods to initialize encoders and count wheel ticks.

MotorControl.py

Contains methods to calibrate speeds and set speeds of motors with pwm and ips.

Orientation.py

Contains methods to ask user for degrees and seconds, and performs robot rotation.

Pid_control.py

Contains various proportional control functions to dictate the speed, and reports saturated IPS.

Task1.py

Contains functions that control the robot, allowing it to drive forward until it reaches a dead-end, tracking the distances as it journeys.

Task2.py

Contains functions that allows the robot to travel forward while maintaining a parallel trajectory relative to the side-walls, employing proportional control to determine speed.

Task3.py

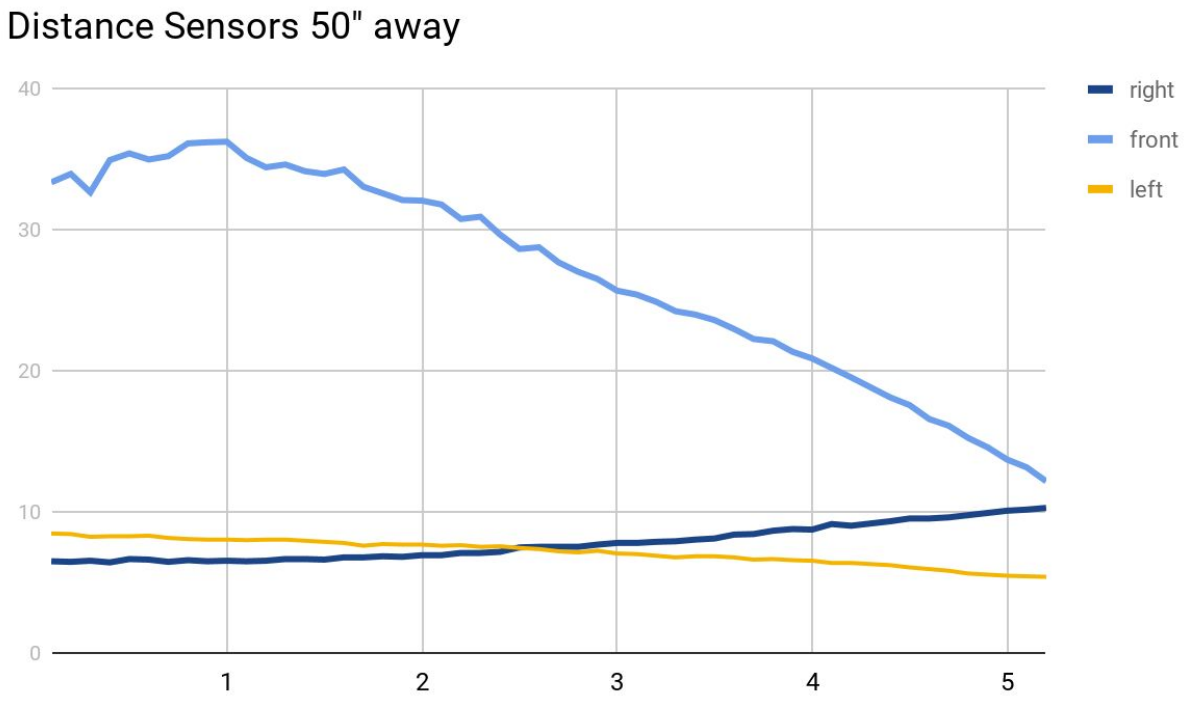
Contains methods to allow the robot to follow the wall to navigate through the maze. Prompts the user to input a side before navigating through the maze, following the indicated wall-side.

Tof.py

Contains various functions to initialize the sensors, and allows for recording and returning sensor data.

SECTION 1

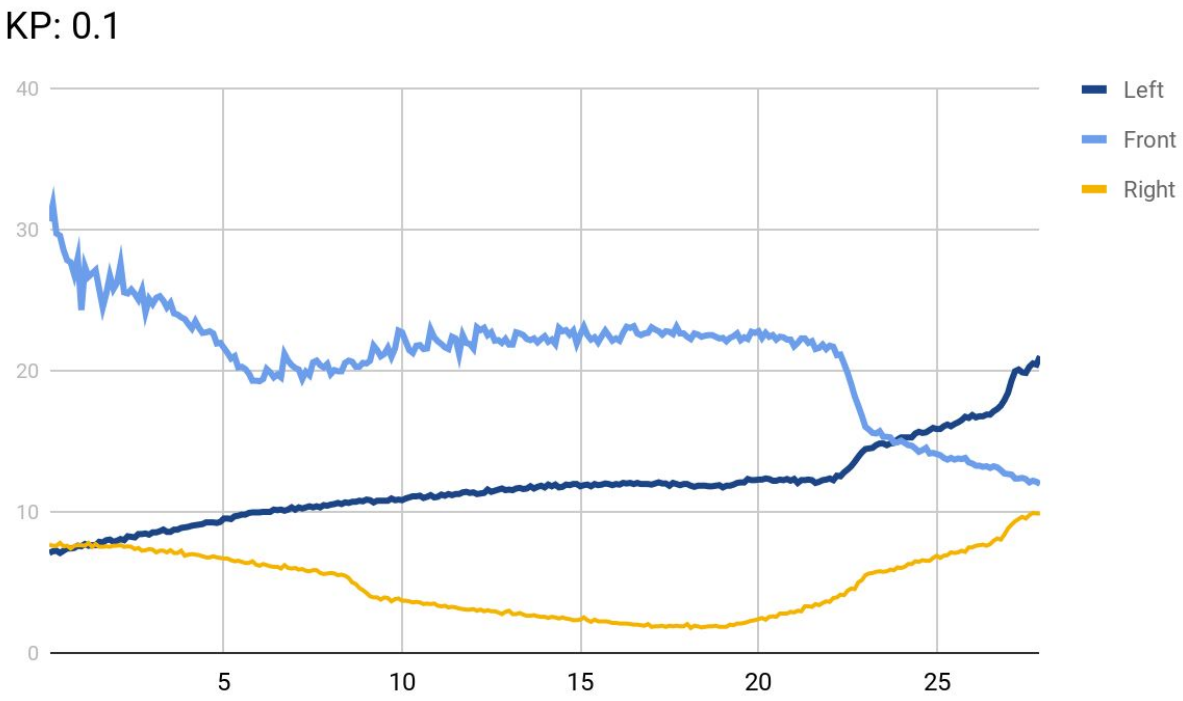
Task 1 Chart



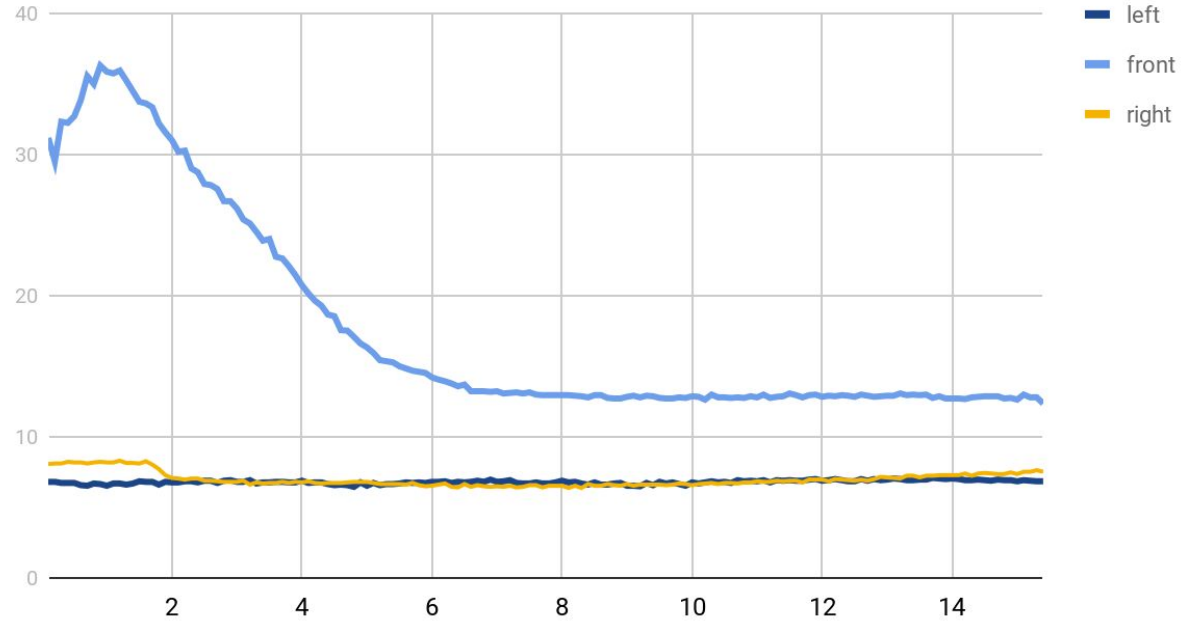
The robot's velocity was set to 5 inches per second through use of SetSpeedsIPS, and the velocity remained unchanged for the entire journey. Using the front sensor, it kept track of the distance between it and the wall in front of it -- so long as the wall was further away than the user-inputted value, the robot would continue to move forward. When the wall was closer than the inputted value, the robot would come to a full stop. As you can see in the chart above, the robot began to veer closer and closer to the left wall as it moved forward -- this issue was caused due to a mild variation in wheel speeds despite sharing the same input. One wheel moved a bit slower, causing the robot to veer towards the left wall.

SECTION 2

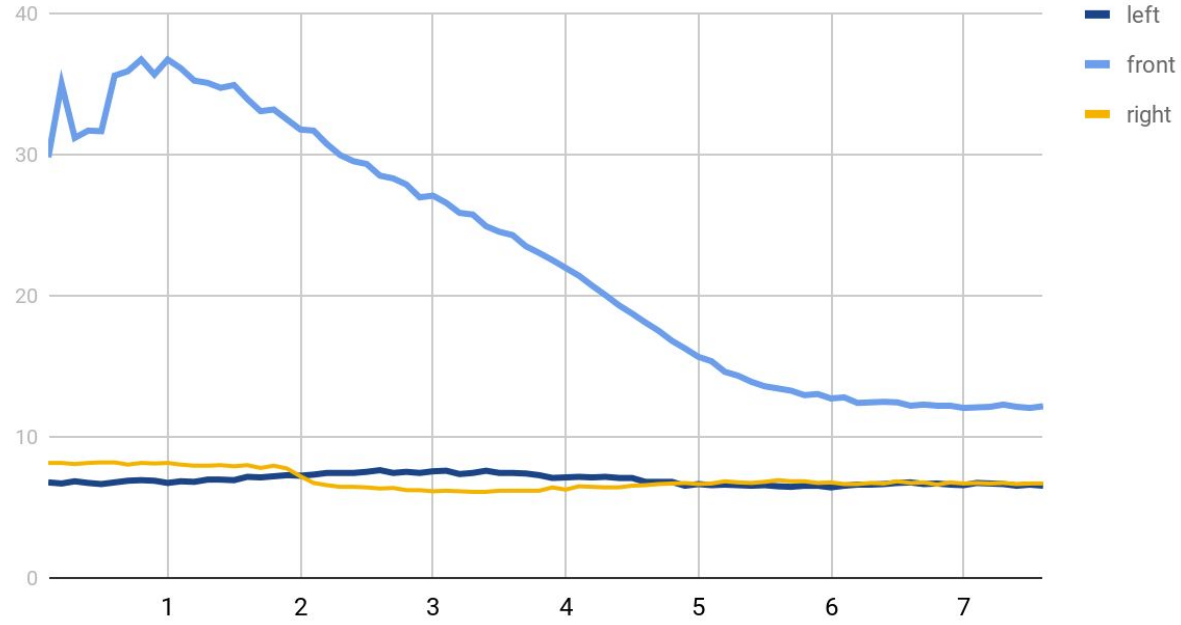
Task 2 Proportional Control Chart



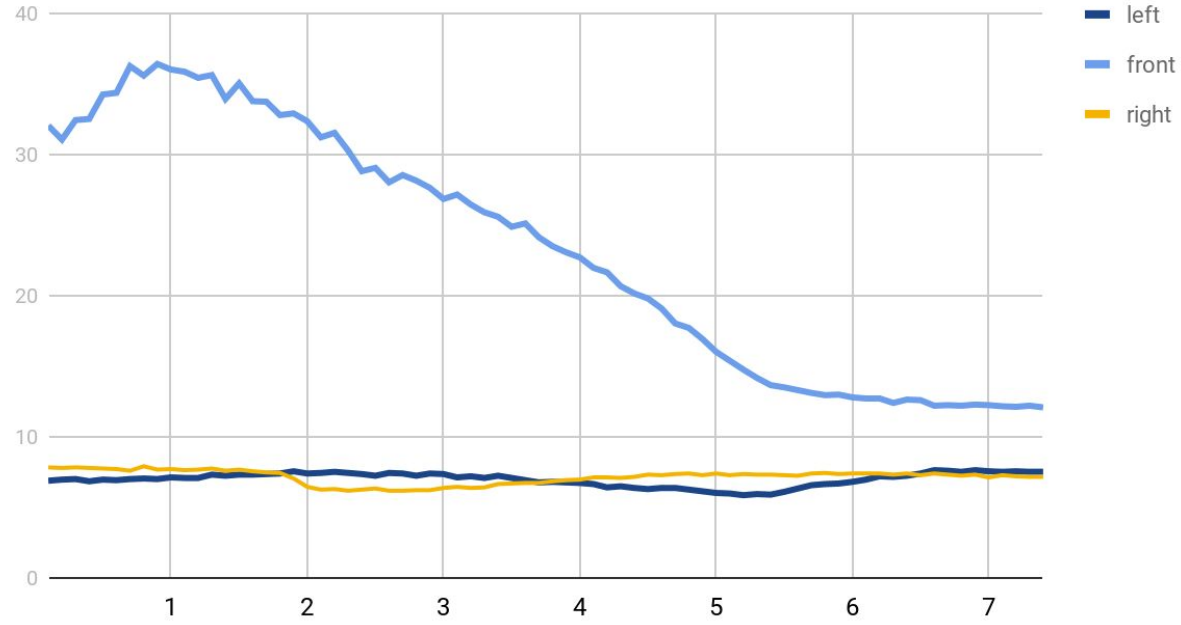
KP: 0.5


40
30
20
10
0
2
4
6
8
10
12
14
left
front
right

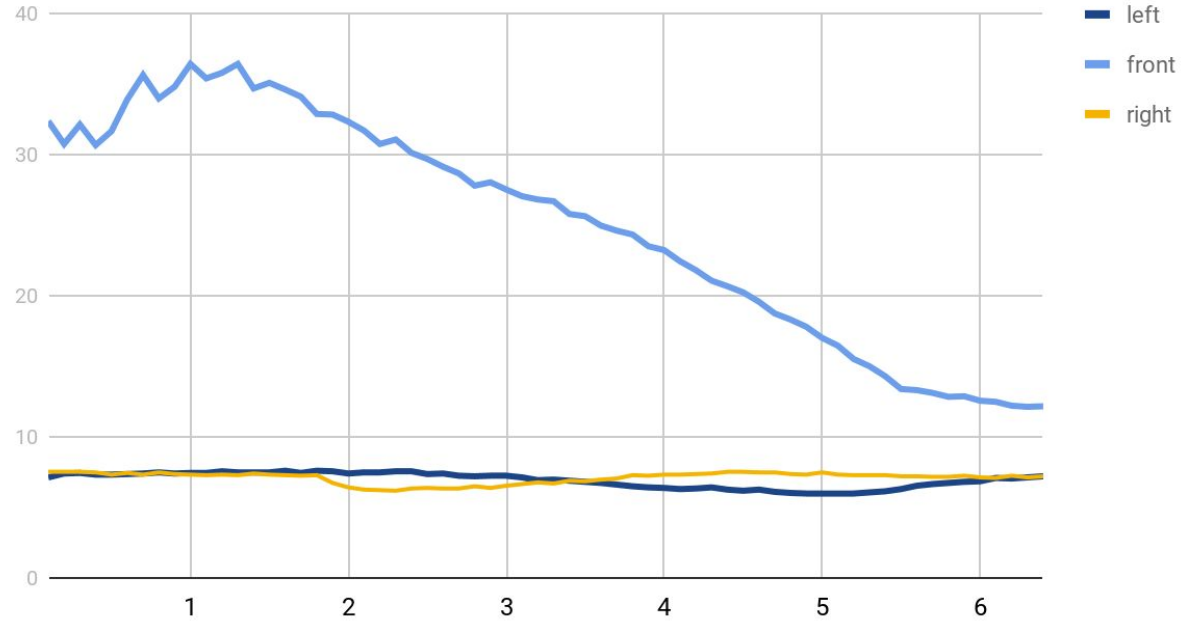
KP: 1.0

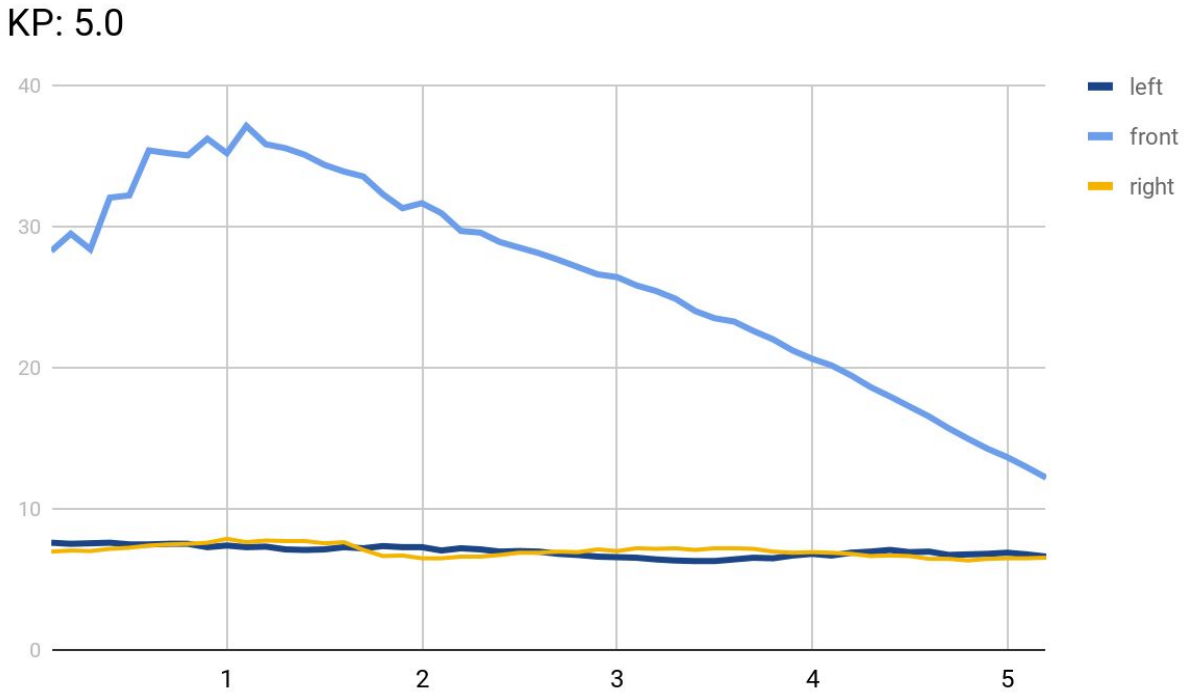

40
30
20
10
0
1
2
3
4
5
6
7
left
front
right

KP: 1.5


40
30
20
10
0
1
2
3
4
5
6
7
left
front
right

KP: 2.0


40
30
20
10
0
1
2
3
4
5
6
left
front
right



Important Variables and Functions:

-leftActualDistance = tof.getLeftDistance()

-rightActualDistance = tof.getRightDistance()

-frontActualDistance = tof.getFrontDistance()

-desiredDistanceForFront = 12 {Hardcoded value}

-desiredDistanceForSides = (leftActualDistance + rightActualDistance)/2

-forwardIPS = pid.getDesiredSpeed(desiredDistanceForFront, frontActualDistance)

-lIPS = rIPS = forwardIPS

-lIPS = lIPS - pid.getDesiredSpeed(desiredDistanceForSides, leftActualDistance)

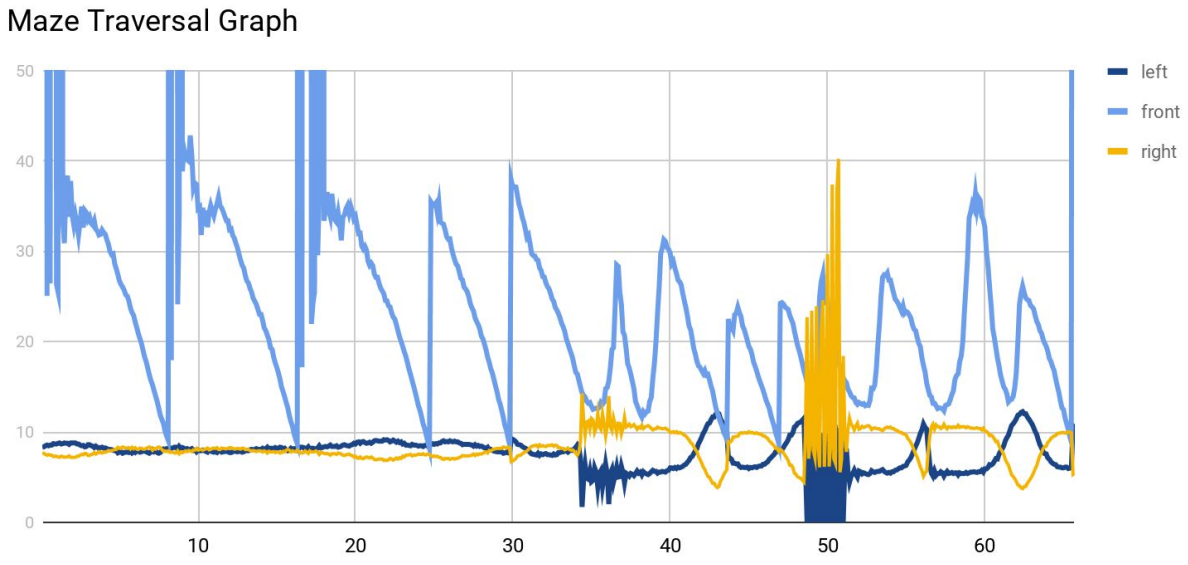
-rIPS = rIPS - pid.getDesiredSpeed(desiredDistanceForSides, rightActualDistance)

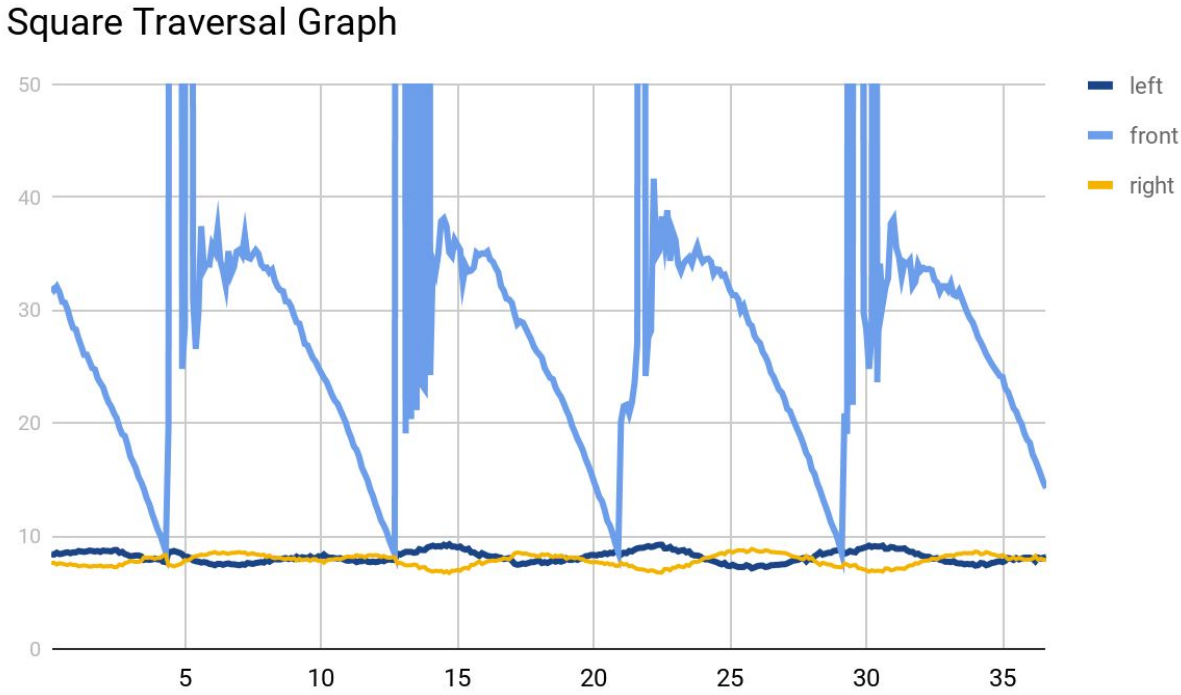
The robot established a *desiredDistanceForSides* by using the left and right sensors to add up and average the distances between the walls. The robot then used that information with a self-updating pid.getDesiredSpeed function to control the wheel's velocity. Each wheel had its velocity independently adjusted. It used this to drive forward while keeping itself parallel to the

walls. When the robot registered a wall less than 12 inches ahead of it, it came to a full stop, ending the task.

SECTION 3

Task 3 - Wall Following Charts





The first thing to note is that Task 3 contains all of the same functions that Task 2 possesses, and all of those functions work the exact same. It also adds a few new functions, which will be elaborated on in this section.

Important (New) Functions and Variable:

-*wallSide*: the user inputs which side to follow, “right” or “left”, which is used in the following two functions below.

-*decideToRotateLeftRightOrUTurn*(tof, orientation, wallSide):

if wallSide = left, the robot will prioritize turning left when possible (rotateDegreesAtMaxSpeed(-90)), right when left is not available (rotateDegreesAtMaxSpeed(90)), and U-Turn when hitting a true dead end (rotateDegreesAtMaxSpeed(180)).

if wallSide = right, the robot will prioritize turning right when possible, left when right is not available, and it will only U-Turn when hitting a true dead end.

-getSaturatedSideDistances(tof, desiredDistance, wallSide):

If wallSide = right, the function will return the actual right-side distance as reported by the sensor, and the left-side distance will be the “desiredDistance” subtracted by the difference between the “desiredDistance” and the rightActualDistance.

If wallSide = left, then the above is the same except using leftActualDistances when appropriate rather than right-side distances.

The program will initially prompt the user to input which wall to follow: left or right? The decision will ultimately guide the *decideToRotateLeftRightOrUTurn* function.

As the robot continues to follow the path in the maze, when it encounters a gap without a dead-end, if the gap is on the side it’s intended to be following, it will curve around and continue to follow the wall. This provides a rudimentary navigational method for it. When it encounters a dead end, it will consult the decision function to determine which direction to turn, and continue on its way. Provided the entire maze can be navigated, the robot will eventually make its way through the entire maze.

SECTION 4

https://www.youtube.com/watch?v=e8hfrkMoFhs

SECTION 5

This project had its fair share of dilemmas, most of which stemmed from the front sensor specifically. The sensor had issues with the end wall when it was beyond a certain range, defaulting to an extremely low value and prompting the robot to back up thinking it was too close to the wall. This was circumvented in two different ways.

The first and most important method, done through coding, instructed the robot that if the sensor reported a front wall in less than three inches, it would regard it as an arbitrarily long distance. While not a perfect fix, it worked because generally the robot should typically have stopped at around ten to twelve inches away from the wall, so if it reports the wall being under three inches ahead that most likely means it's actually very far away.

The other method, primarily used to verify that the wall following worked properly, involved creating an artificial wall in front of the robot, tricking the robot into thinking the wall wasn't infinite distance away (and thus not under three inches away, prompting it to back up). Just keeping a box a set distance away from the sensor at all times was enough to fulfill this need. Ideally, this should not have been necessary, but the sensor problems didn't truly end with the replacement -- instead of not working, it instead only worked so far, and the corridors were too long for the device to properly track.

The first task was easily completed, relatively speaking, as it merely required the robot to drive forward until the wall was about twelve inches away. Proportional control was not required, nor was keeping the robot parallel to the side walls. Thus, this task was completed without issue.

The second task caused significantly more issues, as the hurdle of keeping the robot parallel was difficult to overcome… at first. The solution, we found, was to maintain a reliance on the proportional control aspect of the forward drive while applying a modifier to the speed based on how close or far the walls were from the robot, with the robot effectively adjusting its

speed on a wheel-per-wheel basis to keep the walls about the same distance away from it (and thus, parallel).

The third task caused the most issues by far, as the robot had an unfortunate habit of getting stuck while turning certain corners. It would see the front wall as too close, then too far, all while trying to stick close to the wall it needed to follow; this prompted the robot to simulate suffering a seizure, and that's not an exaggeration or a joke, it quite literally shook in place while trying to move in multiple directions simultaneously. Ultimately, the solution involved emulating the previous task with a modification -- rather than use the literal distance from the walls, a "saturated" distance was devised using a separate function, based on the wall-side that was to be followed. When the robot hit a dead end, it would then check the wall it needed to follow, scan the local area, and use these data points to decide if it should turn left, right, or turn around entirely.