Introduction

This project required us to navigate a maze in both a Gazebo simulation and in real life using TurtleBot3 models. Our algorithm was initially devised in the simulation and was then incorporated into the real-life testing. The problem posed by this project required the usage of navigation, LiDAR, simulation, and Python skills learned throughout the semester. Most relevant for this group was homework assignment 5, as the basic turning and alignment algorithms translated easily to this project. While initially using homework 5, the group departed from wall-following into new logic sequences, a key factor in our success in both the hardware and software aspects of the project.

Hardware

For the initial approach to hardware testing, wall detection, distances, and turning were first explored. Using work from Homework 5, we implemented simple turning algorithms (from the Inside Wall following). The robot moves forward until it detects a wall at which point it turns to the right. Additionally, the robot could automatically correct the distance between it and the wall, to avoid overshooting. While this was useful, its inability to handle variations in turn direction made fine-tuning ineffective. Additionally, specifically in the hardware realm, it seemed that a simple line-following algorithm would be inefficient in speed, and a new logic would be beneficial to devise.

While the decided algorithm is later discussed in the simulation section, our most important priority was detection. The turtlebot3 had extremely unreliable sensing, which was much more unpredictable when contrasted with the gazebo simulation. Originally, this issue was solved by slowing down the process; turning updates was slower than the gazebo simulation. However, it was found that a more important change would be a change in approach: Instead of comparing values in specific directions, it became easier to add up the collected values over a range of directions of the LiDAR.

The most significant challenge to achieving hardware success, after completing the simulation maze, was the translation of the algorithm from gazebo conventions to the hardware conventions, especially with angles. Angle conventions in both Gazebo and on hardware were similar but difficult to integrate in many ways.

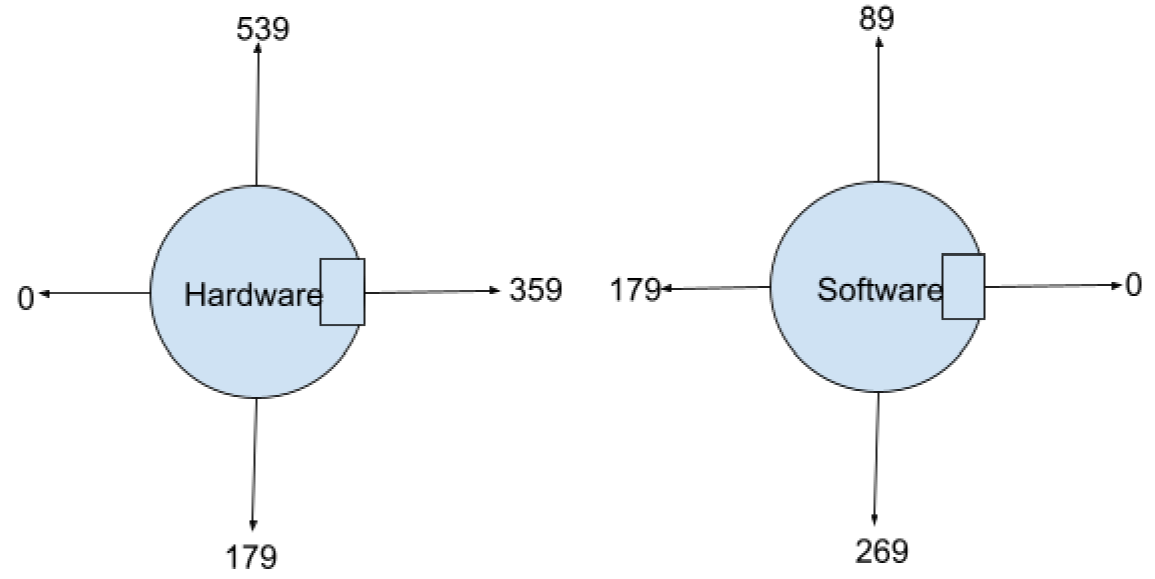
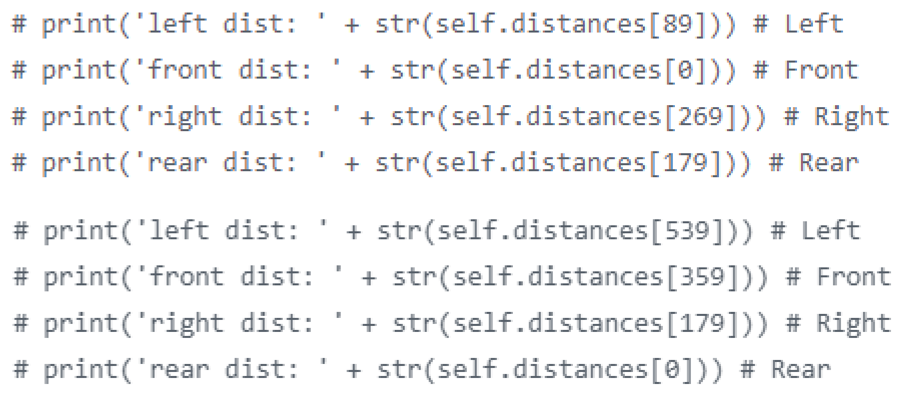


Figure: *Angle conventions of hardware and software.*



```
# print('left dist: ' + str(self.distances[89])) # Left
# print('front dist: ' + str(self.distances[0])) # Front
# print('right dist: ' + str(self.distances[269])) # Right
# print('rear dist: ' + str(self.distances[179])) # Rear

# print('left dist: ' + str(self.distances[539])) # Left
# print('front dist: ' + str(self.distances[359])) # Front
# print('right dist: ' + str(self.distances[179])) # Right
# print('rear dist: ' + str(self.distances[0])) # Rear
```

Code Block: *Comparison of direction statements to show angle conversions*

After adaptation of the angles, and navigational fine-tuning the disparity between our robot performance and the original right wall following became more apparent. On the day of the competition, while testing in lab hours we initially achieved comparable results to other groups using wall-following, with a minimal time advantage that we saw as appropriate. As the figure below shows, our algorithm mainly avoids certain redundancies that right-wall following befalls, specifically in the pockets circled in green. During testing, this is something we noticed, but since the practice maze looked like the left half of the maze below, it's easy to see that the speed advantage of our algorithm was generally minimal. However, in the final testing, the maze construction allowed for several diagonal paths (as demonstrated on the right side of the figure below) which drastically cut down solving & navigation times for our algorithm.

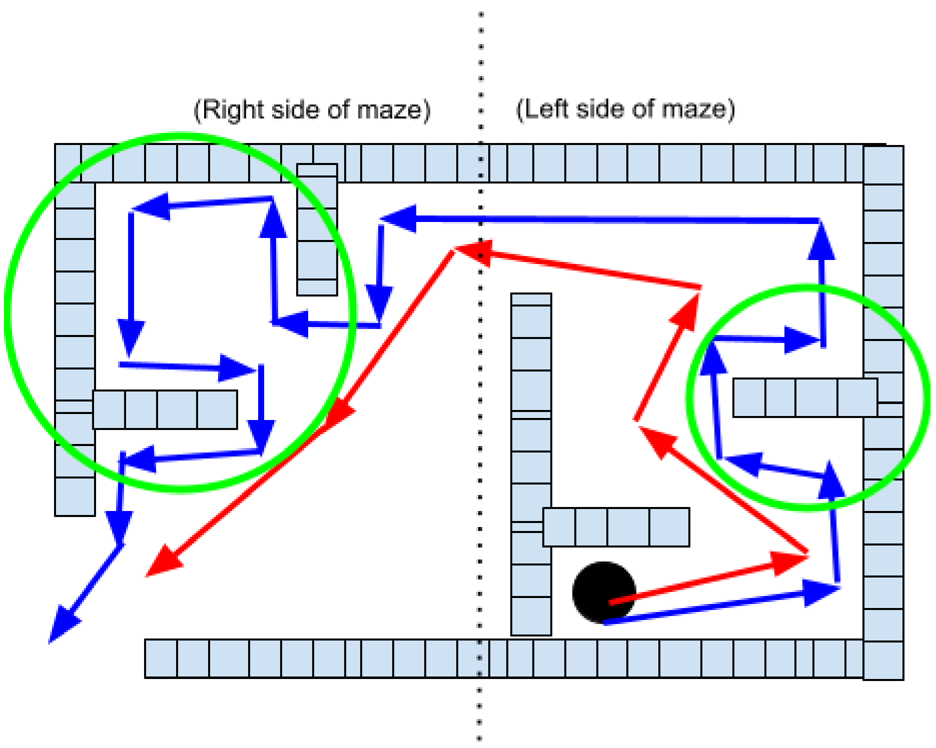


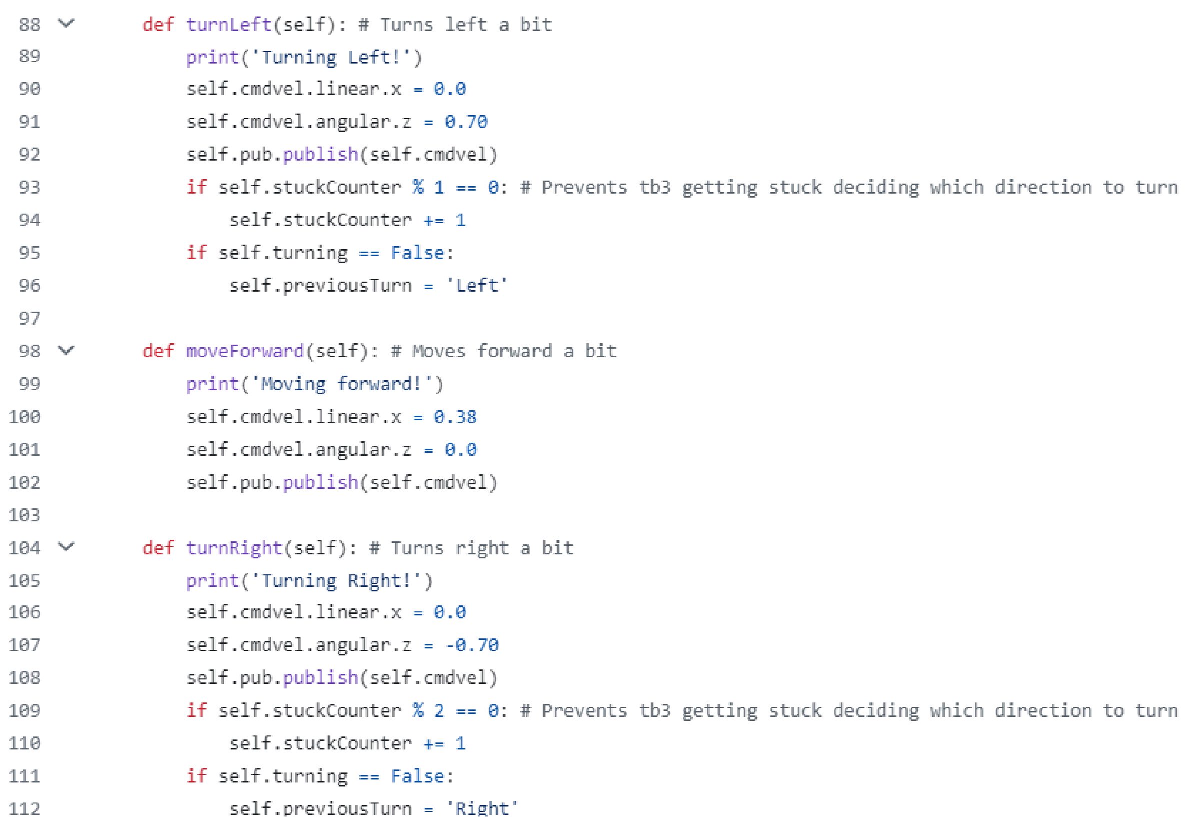
Figure: *Maze Comparison of our algorithm (Red) to generic right-wall following (Blue)*

In competition, our team achieved a time of 00:28, drastically lower than the average 2-3 minute range, specifically because of the advantages that diagonal navigation afforded in that specific map. The trade-off to this algorithm switch was reliability, to some extent. While not a huge discrepancy, our robot was much more likely to get "confused" about the direction it should move in, as discussed below. From a results perspective, it made the timing of our runs vary more greatly, meaning that we sacrificed some precision for greater accuracy on some runs. This, in a competitive environment, benefitted us greatly, and even our least accurate runs managed to compete with our contemporaries' maze-solving speeds.

Simulation

Initially, the method used to solve the maze was a simple right-wall following algorithm, in which the robot turns to the left when it encounters a corner with a wall ahead of it and to its right. This algorithm evolved so that the robot would check what sides the wall was on. If the robot detected a wall on the left and in front of it, it would turn to the right. If the robot detected a wall on the right and in front of it, it would turn to the left. This, while simplistic, is an extremely effective algorithm to solve the maze. To further speed up the robot's solving skills, a more situation-based approach was necessary.

First, the most basic of the robot's turning skills were implemented. Turning was something that needed a lot of troubleshooting, so a "stuckCounter" was created to prevent the tb3 from failing to decide on a direction to turn. Additionally, the algorithm kept a memory of the most recent direction in which it turned. These turns were generally small increments so that the robot could collect a variety of data as it “wiggled” to see the distances on its left and right. Ultimately, most of the turning remains the same as in previous assignments.



```python
    def turnLeft(self): # Turns left a bit
        print('Turning Left!')
        self.cmdvel.linear.x = 0.0
        self.cmdvel.angular.z = 0.70
        self.pub.publish(self.cmdvel)
        if self.stuckCounter % 1 == 0: # Prevents tb3 getting stuck deciding which direction to turn
            self.stuckCounter += 1
        if self.turning == False:
            self.previousTurn = 'Left'

    def moveForward(self): # Moves forward a bit
        print('Moving forward!')
        self.cmdvel.linear.x = 0.38
        self.cmdvel.angular.z = 0.0
        self.pub.publish(self.cmdvel)

    def turnRight(self): # Turns right a bit
        print('Turning Right!')
        self.cmdvel.linear.x = 0.0
        self.cmdvel.angular.z = -0.70
        self.pub.publish(self.cmdvel)
        if self.stuckCounter % 2 == 0: # Prevents tb3 getting stuck deciding which direction to turn
            self.stuckCounter += 1
        if self.turning == False:
            self.previousTurn = 'Right'
```

Code Block: *Turning of the Robot (Both simulation and hardware)*

For the most attainable yet more efficient product, we decided to keep it simple. We came up with a simple distance comparison algorithm, and combined it with a wall-avoiding procedure to avoid any collisions with the robot.

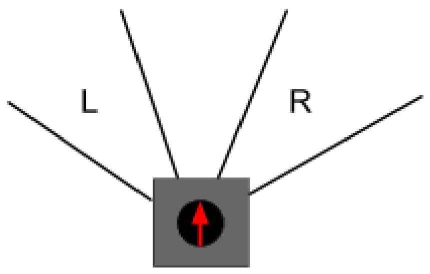
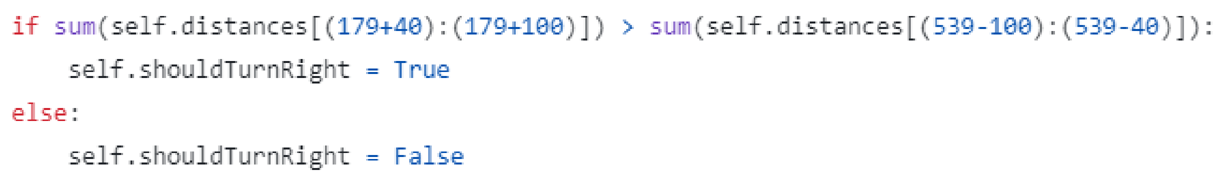
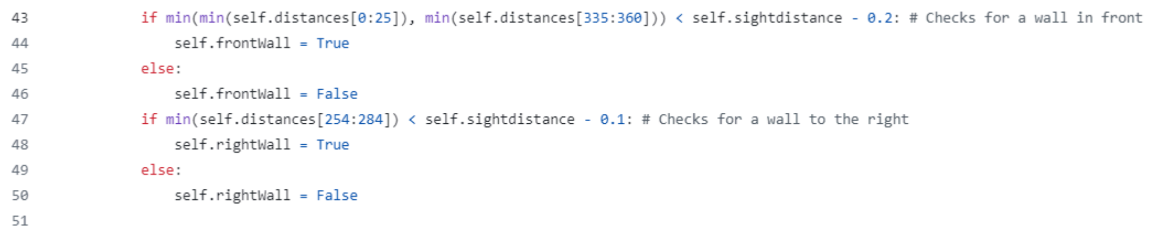


Figure: *Simple Diagram of the TB3 with “Left” & “Right” Projections*



```python
if sum(self.distances[(179+40):(179+100)]) > sum(self.distances[(539-100):(539-40)]):
    self.shouldTurnRight = True
else:
    self.shouldTurnRight = False
```

Code Block: *If-Else Loop that explicates the logic described below.*

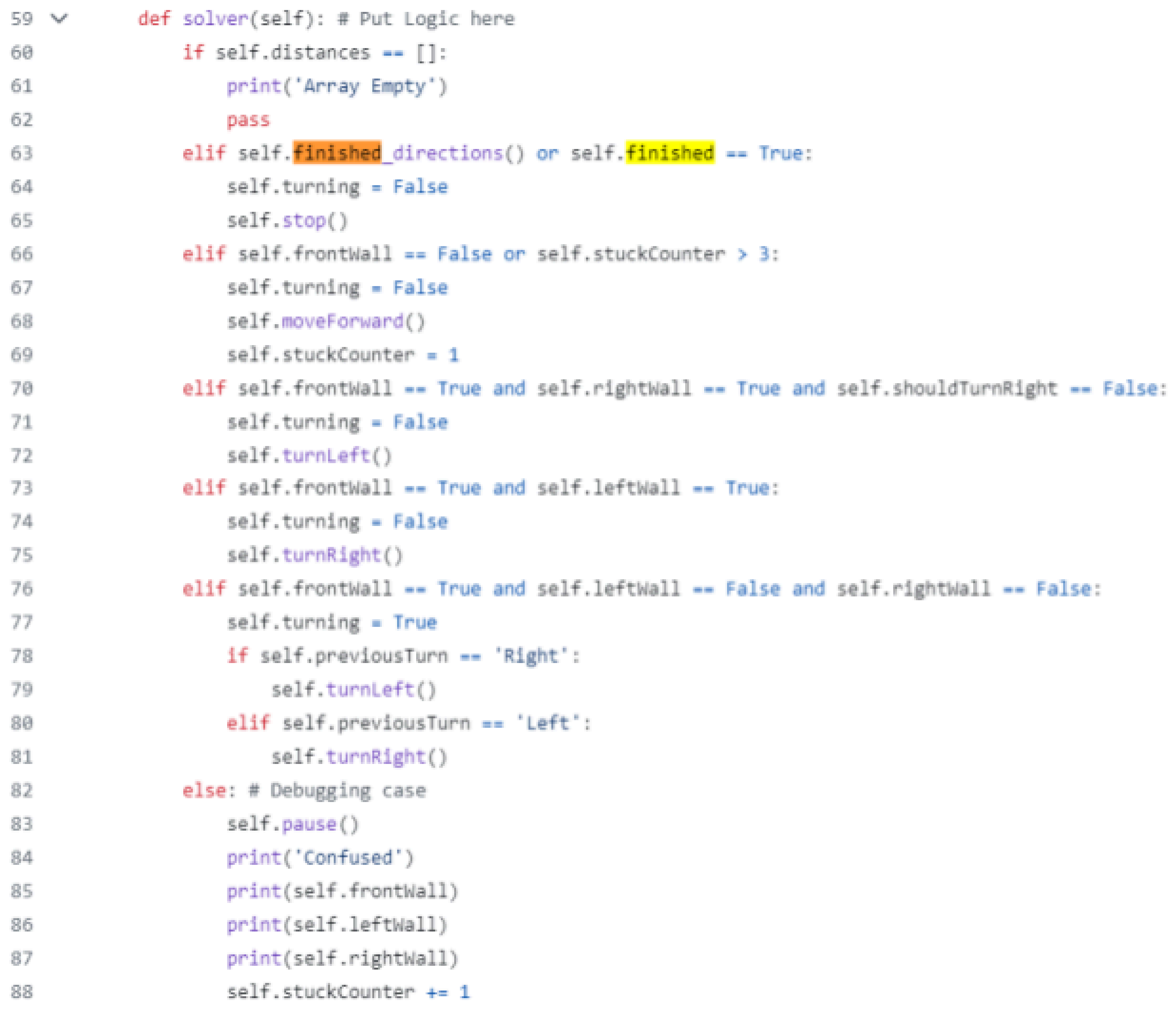


```python
        if min(min(self.distances[0:25]), min(self.distances[335:360])) < self.sightdistance - 0.2: # Checks for a wall in front
            self.frontWall = True
        else:
            self.frontWall = False
        if min(self.distances[254:284]) < self.sightdistance - 0.1: # Checks for a wall to the right
            self.rightWall = True
        else:
            self.rightWall = False

```

Code Block: *Wall Detection Code*

In the final algorithm we used, the robot (pictured above) sums all the distances between certain ranges on the left ‘L’ and right ‘R’ sides of the robot. The side with a higher total distance is the direction in which the robot moves. This simple situation-based thinking allows the robot to improve its navigational skills, Additionally, since this method does not follow a wall, as discussed in the hardware section of this report, it reduces the existing redundancies in certain pockets of the maze. While our logic relied primarily on this, we did retain some wall-based logic to make a more efficient navigation system. If the robot detected a wall in front of it, and to either side, it was told to turn away from the wall, to streamline cases when the robot was at a corner in the maze. Additionally, (as evident in Lines 76 through 81) the algorithm also stored previous turn information, so that if the robot does not detect a wall on either side but does in front of it, it turns in the direction opposite of its last turn.



```python
    def solver(self): # Put Logic here
        if self.distances == []:
            print('Array Empty')
            pass
        elif self.finished_directions() or self.finished == True:
            self.turning = False
            self.stop()
        elif self.frontWall == False or self.stuckCounter > 3:
            self.turning = False
            self.moveForward()
            self.stuckCounter = 1
        elif self.frontWall == True and self.rightWall == True and self.shouldTurnRight == False:
            self.turning = False
            self.turnLeft()
        elif self.frontWall == True and self.leftWall == True:
            self.turning = False
            self.turnRight()
        elif self.frontWall == True and self.leftWall == False and self.rightWall == False:
            self.turning = True
            if self.previousTurn == 'Right':
                self.turnLeft()
            elif self.previousTurn == 'Left':
                self.turnRight()
        else: # Debugging case
            self.pause()
            print('Confused')
            print(self.frontWall)
            print(self.leftWall)
            print(self.rightWall)
            self.stuckCounter += 1
```

Code Block: *Main Solver algorithm used in both Hardware and Software*

The biggest issue with the main procedure is a theoretical fork in the maze. There exists a small risk that the robot would choose the wrong path and end up going backward/in a direction away from the exit. However, practically, we found that this only adds to the solution time, and the robot always eventually exits the maze.

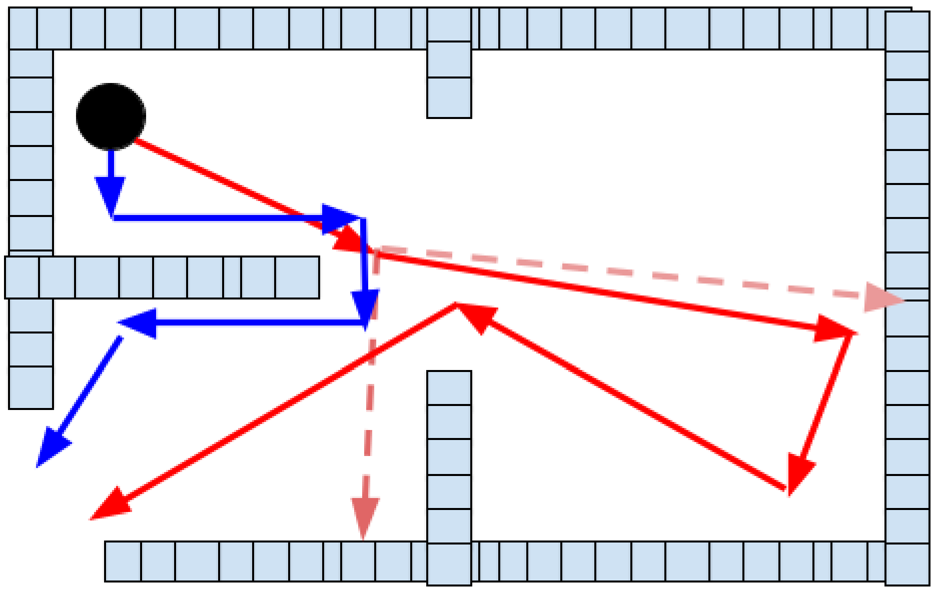
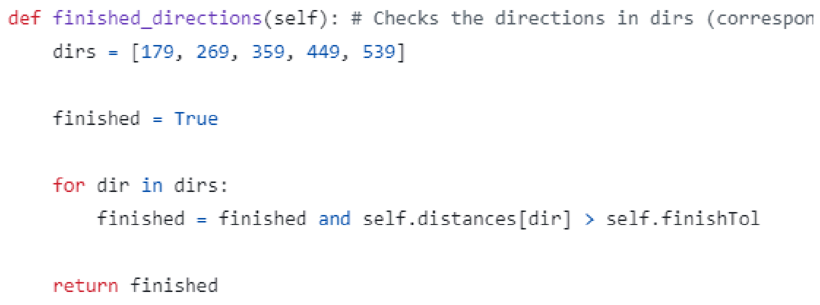


Figure: *Theoretical Maze set up where this algorithm is beat*

As the maze above demonstrates, even if the robot were to make a "wrong" turn in the maze, it would eventually return to the point of the "wrong decision", and, after enough iterations of scanning this particular spot, would eventually exit the maze. In the example above, the two dotted red arrows try to represent the two measurements which makes the robot turn to the left instead of towards the exit. This makes apparent the application-based disadvantages that this algorithm can have when compared, again, to the wall-following method.

For the exit clause in software, (regarding when the maze has been exited), the program was exceedingly simple. The robot is to check the distances around the robot in all directions. This tolerance was found experimentally.



```python
def finished_directions(self): # Checks the directions in dirs (correspon
    dirs = [179, 269, 359, 449, 539]

    finished = True

    for dir in dirs:
        finished = finished and self.distances[dir] > self.finishTol

    return finished
```

Code Block: *Exit Clause for the maze*

Conclusion

The most successful ethos in this project was that simple was better: the easy-to-understand mathematical functions were simple to model onto the TurtleBot. Even our more “complicated” logic was the most simple devisable way to solve a maze. This worked in our favor, as, instead of trying to make a complicated logic work consistently, we could spend our time perfecting the variables surrounding the simple procedure which we knew would be successful in solving the mazes.

In retrospect, too much time was invested in fine-tuning, where larger changes would have been a more productive effort. This was especially important when we figured out that our specific numbers were not affecting our hardware attempts, but deeper issues regarding data collection would have been better to explore. Our group saw the largest improvement when we moved from a specific instance-based code to algorithms which used ranges of data to make decisions since the LiDAR would not always necessarily be reliable in any single dimension.

Ultimately, we learned this lesson during this project, and the only improvement that could have been made was realizing this sooner, so we could spend more time adding and testing new ideas to our robot’s runs. Our simple approach benefitted us greatly, but greater experimentation with different simplistic approaches would have benefitted us more.

Team Accountability

Shanay spent most of his time understanding and improving the logic in Gazebo and devised the more successful algorithm we ended up using. Trevor was responsible for the in-person robot testing and understanding of the code transition from Gazebo to the TurtleBot, while Aryaman worked on formatting, recording, and creating the report.