Lab 2 Report

Mathematical Computations :

Task 1:

Task 1 involved my function pid_wall_stop(self, leftOrRightSide, targetDistanceWall, targetDistanceSide, proportionalGainWall, proportionalGainSide):

LeftOrRightSide: Specified which wall to follow. “left” or “right”.

TargetDistanceWall: The desired distance the robot should stop at with its front sensor

TargetDistanceSide: The desired distance the robot should maintain within the side walls

ProportionalGainWall: the k value for the change of velocity on the robot

ProportionalGainSide: the k value for maintaining within the side walls

Front distance error (aka how far or close the robot is to the desired value) was calculated as **–targetDistanceWall - (-lidar_front_sensor_reading)**. The negative signs were put to receive the correct sign for the errors and future saturation functions.

Similar errors (errorSideL and errorSideR) for the left and right sensors on the robot were calculated as **(lidar_left_or_right_sensor_reading – targetDistanceSide).**

Our control variable for our saturation functions is the **front distance error * proportionalGainWall.**

The control variable determines what the velocity variable is set to based on these saturation functions:

// max_vel = 20 and min_vel = -20

```
if control > max_vel: // if control is high, make robot go to max_velocity
velocity = max_vel
elif min_vel <= control <= max_vel: // if control is in the middle, control = velocity
velocity = control
else: // if control is lower than min, keep it at the min_vel
velocity = min_vel
```

After the velocity is determined, we now must check if the robot is close to a wall.

Depending on either our “left” or “right” parameter, if our errorSideR or errorSideL is > 0, this means that the robot is further away from the desired distance with the wall, thus it will turn the vehicle left or right accordingly as follows:

```
self.pid_turn_left_or_right(velocity - (abs(errorSideL or errorSideR) * proportionalGainSide), velocity)
```

This function calls a simple execution where left and right motors are changed accordingly based on the saturated function’s velocity and the lesser velocity calculated as velocity – a side error control which is the absolute value of the side error * our k value for side.

Task 2:

```
def pid_around_wall(self, proportional_gain, velocity, distance_from_wall, left_or_right):
```

For Task 2, I created a new Proportional Controller function

Proportional_gain: k value for velocity change

Velocity: desired velocity value

Distance_from_wall: desired distance away from the wall the robot is following

Left_or_right: specify “left” or “right” wall following.

First, our lidar sensor collection for the left or right walls are collected as an average. For example, if we chose right following, our averaged_sensor_reading would be the following:

```
average_range_image = mean([

self.get_lidar_range_image()[525],

self.get_lidar_range_image()[550],

self.get_lidar_range_image()[600],

self.get_lidar_range_image()[575]

])
```

These functions collect the lidar distance readings from the most rightward position of the lidar sensor to values above it, in order to ensure complete accuracy with the error value.

Error will be distance_from_wall – average_range_image.

Control wll be the absolute value of error * our proportional_gain.

The following is my saturation functions: (saturation function for left and right are the same with different executions, but I will show the right wall version).

```
if self.get_lidar_range_image()[400] < 0.3: // first checks if the robot is at risk of hitting the wall
```

```
self.rotate_certain(90, 2, "left") // calls a function that will rotate the robot 90 degrees to the left at 2m/s.

elif -50 < error < -0.8: // if the robot is between values that indicate there is no longer a wall...

self.circular_rotation(math.pi/2, .5, velocity, "right") // do a 90 degree rotation to the right

elif error > 0: // if the robot is further away then the desired value to the wall

self.pid_turn_left(velocity - control, velocity) // turn left with the lesser velocity being the initial velocity - control

else: // if the robot is closer than the desired value to the wall

self.pid_turn_right(velocity - control, velocity) // turn right with the lesser velocity being the initial velocity – control
```

The computations of Task 1 and Task 2 are encompassed by a while loop to have them constantly done each time step.

Conclusions:

Starting with Task 1, it was not clear to me that a second proportional control was needed to ensure that the robot would not hit the walls. I struggled on that aspect the hardest, as i did not know how to implement the additional feature with my already prexisting saturation function. I realized that I had to break my code into two parts: a saturation function for obtaining the velocity, and then a saturation function that would turn left or right given conditions, and that the lesser velocity would be the normal velocity – the control for my side distance. For the base case, where the robot is exactly in the middle, which is desired, the task is completed normally. For my other test cases, while it would never perfectly reach the middle again, it would do a good enough job and stop at the desired distance, given the fact that the robot's controller does not include the ID in PID.

With Task 2, I had an initial draft function which I scraped because it did not work properly, and restarted from scratch thinking about how the error could define the movement. The biggest problem I ran into was with a single lidar sensor value determining the robots movement. In a perfect world with good turns, one value should be enough, but because the robot will inherently sway, values for the saturation function would be thrown off completely by wrong measurements, so to counteract this I had my distance sensor to the wall be an averaged value of four sensor measurements, starting from the right or left most, and then increasing its range by getting values above them. This change ensured that the turns were done properly. I no longer had issues making 180 degree turns, however, the robot is not fast enough to have sharp turns. In the instructions, it says that the front distance sensor is not required, but it does not say it cant be used. If the robot gets too close to the wall with its front, the saturation function instructs it to make a 90 degree turn left or right. I had to make a new rotate function rotate_certain(self, target_rotation, vel, LoR) that would get the accumulated amount of degrees the robot turns and then stop once it reaches the target_rotation. After these changes, I was able to successfully get my robot to follow the wall.