Lab Report 3

By Joshua-Luke Lacsamana

Task 1:

The following function is used for motion to goal: `def motion_to_goal(self, velocity, K_object, K_curve):`

Where velocity is velocity, k_object is the proportional gain for the velocity, and K_curve is proportional gain for the left and right movement.

The mathematics of this function is very similar to the one used in lab 2 task 1, where the robot needs to stop at a certain distance from the wall using pid. In this case, we first use data from our virtual camera to detect a solid colored object. If there is no object visible, the robot will turn in the correct direction until it is found. I will elaborate on "correct direction" in a second.

"For obj in objects" will grab the object that is detected in the objects class, in our case, there is only one object to be seen.

Our velocity error is calculated as `error = (-0.5) - (-position_on_image[0]).`

Where –0.5 represents the distance we want away from the wall and position_on_image[0] returning the webots calculated distance from the camera to the object.

Another error we need to consider is the error for our saturation function to determine its left and right movements whenever the object moves. Named "turnError", it is defined as

```
turnError = position_on_image[1]
```

Position_on_image[1] returns a positive or negative number that determines the orientation of the robot with the object. For example, if the value is 0, that means the robot is perfectly straight with the object, if the value is positive, the robot will need to turn left, if the value is negative, turn right.

```
control = error * K_object

if control > max_vel:

velocity = max_vel

elif min_vel <= control <= max_vel:

velocity = control

else:

velocity = min_vel
```

This code defines our saturation function to determine the speed of our robots velocity, it is a similar approach to the previous lab. Max_vel and min_vel are 20 and –20, respectively.

```
if turnError > 0:

self.pid_turn_left(velocity - (abs(turnError) * K_curve), velocity)

else:

self.pid_turn_right(velocity - (abs(turnError) * K_curve), velocity)

if ( 0.0 <= control <= 0.080):

print("Program should now stop")

self.stop()
```

This saturation function determines the movement of the robot based on the turnError. The last if statement handles the case if the control value reaches between those values, it means the robot has encountered the object and no longer needs to keep moving.

The pid_turn_leftorright() function just activates the turning feature, just like lab 2.

Task 2:

Task 2 involves a combination of the motion_to_goal() function included with the pid_around_wall() function I created for lab 2. Modifications were needed to account for different obstacles the robot would encounter.

```
bug_0(self, velocity, K_object, K_curve, rotating_degree, distance_from_wall, proportional_gain):
```

This function takes in parameters for the previous functions mentioned. A new rotation function was created, and rotating_degree() defines not only the degree of rotation when encountering a wall head on, but also determines whether the wall following will be left or right.

As for the mathematical components, they are a combination of wall following and motion_to_goal(), which were both already elaborated fully. However, the functions are controlled through states. There are if statements placed through these blocks of code that will switch what segment of the bug algorithm will play out depending on the robots circumstances.

Conclussions:

The main difficulty with this lab was figuring out the state changing for the bug 0 algorithm. In my motion_to_goal() function, the robot would move around and turn until it found the object again. However, in Task 2, the robot would get stuck endlessly trying to find the object, as it is being covered by obstacles. Instead, I realized instead that the robot will need to keep moving until it reaches a certain front lidar distance, which will make it rotate 90 degrees and then begin to follow the wall. Also, because the robot will never completley avoid the wall after its 90-degree rotation projection, I added code that will return the min value of a range of left and right lidar sensors to ensure it will not hit the wall, and instead do another 90-degree turn.

A new rotate function was made to account for cases where the current orientation of the robot can switch from 360-0 and 0-360.

```
def rotate(self, degree, vel, margin_of_error=3):

# Calculate the target orientation

target = (self.get_compass_reading() + degree) % 360

# Determine the direction of rotation

if degree > 0:

# clockwise rotation

else:

# counterclockwise
```

The target value is grabbed from the compas reading + degree for turning, and then the remainder is taken from 360. Based on the positive or negative value of the degree, the rotation will go clockwise or counterclockwise.