CDA4621 Lab Report 1 by Joshua-Luke Lacsamana

Here are the following mathematical computations for my move_forward(self, distance, vel) function.

Time=distance/vel, although this may be incorrect due to the fact that velocity is in rad/sec and distance is in meters.

Move_forward() does not do any left and right velocity calculations, since all wheels must share the same velocity in order to move in a straight line.

Here are the following "mathematical computations" for rotate(self, distance, vel) function.

Simply put, in order to rotate the RosBot while having its center body stay on the waypoint, you must negate the pair of the left or right wheel's velocities. Depending on which side you choose to negate, the RosBot will rotate either left or right.

Here are the following "mathematical computations" for circular_rotation(self, circle_angle, circle_radius, max_velocity, left_or_right):

Angular velocity is calculated to adjust the wheel pair's velocity based on a left or right rotation.

v_left(wheels) =ω(R+d_mid) --> this is what the kinematics PowerPoint gives us

This formula assumes that v_left will be the higher velocity, or max, and the notion of v_left or v_right doesn't matter in calculating our angular velocity. If we give a parameter to the function that is to be our max velocity, then we can use this formuka and turn it into the following:

ω(angular velocity) = (max_velocity/(circle_radius + d_mid), where circle_radius is a function parameter as the radius (R) of the ICC we are rotating around, and d_mid is the axil length divided by 2.

We can now get our smaller velocity with:

v_right(wheels) = ω(R - dmid)

Again, although the PowerPoint lists these formulas as velocities for wheels left and right, they are arbitrary, as the code function will be able to choose what wheels to apply the

lower velocity to, causing the circular rotation in either direction. The code will print out the correct value of the left and right linear velocities of the wheels.

The formula turns into: smaller_velocity = angular_velocity*(circle_radius - d_mid)

We can calculate the distance we need to go as distance = (circle_angle*circle_radius), where circle_angle is a function parameter that will range from (pi, pi/2, 2pi, etc.) based on the segment of the circle for circular rotation.

Time is calculated as time = distance/((max_velocity+smaller_velocity)/2). We need to get the average velocity of the two different velocities of the left and right wheels, as that is the true speed of the RosBot.

Based on the whether the RosBot is going left or right, the left or right wheels distance is calculated for our while loop:

Left or right wheel distance = circle_angle*(circle_radius + 0.1325). (If the RosBot goes left, then this would be the calculation for the right wheel's distance.). This distance is used to compare the expected distance of a given wheel and the "in-real-time" wheel displacement of said wheel given the encoders in the motor.

Conclusions:

I started this lab assuming even the essentials such as forward along a distance would be quite simple, but after many trials and errors, there are an enormous number of things to consider when writing functionality for a robot.

I initially was going to use an import of python's Time to stop the RosBots movement after the calculated time given distance and velocity, and that is when I learned the distinction between virtual and real time. I then switched to the idea of recording the encoder distance of one wheel, as its displacement will eventually reach the calculated distance value. This approach worked conceptually, however the RosBot would never truly stop at the exact waypoint position.

At first, I tried tinkering with the distance constant based on dimensions from the RosBot. I even created a new function called chill(self,second) that would give rosbot a break given the second parameter, because I thought it was the execution time that was throwing it off. After an office hour session with Chance, He made me realize that it was just the nature of the program and how robots work. Therefore, I created a margin_of_error() function for my straight and circular rotations, that way it works around

the nature of the while loop not always being perfect in stopping the RosBot on time. However, I will need to do more test runs to figure out the perfect constants for these margin-of-error values, as the waypoints are still not exactly perfect. The margin-of-error constants entirely depend on the inaccuracy of the wheels at different velocities. For a higher velocity, the RosBot is more prone to wheel malfunctions and inaccuracy of encoder measurements, so a higher margin-of-error is needed.

It took me a while to understand the importance of using the wheel encoders, but once I did, I applied formulas from our kinematics lecture and got most of the rotations and straight movements working. Because the rotations have a margin-of-error of having the robot orient slightly off its goal, I used the rotate() function after every turn to orient the RosBot to its desired orientation (it is usually off by 1-2 degrees).

The last big hurdle was the rotation from Point 5 to Point 6. I spent hours rethinking if my equations were incorrect, or if there was a bug in the interface. I failed to realize that doing my own calculations would show that it is physically impossible for the RosBot to squeeze around that corner, which was confirmed by an office hour session. Because of this, I modified the lab slightly by having the movement of Point 4 and 5 go to (-1,1.5,0) -> (1,1.5,0).

In the future, I would like to condense my circular_rotation() function by having the while loop check the distance of all the four motor encoders averaged out, that way I do not need two segments of code that have the same principles with different values.