Lab Report for CISC 3060 Final Lab Report

Fall 2024: December, 16, 2014

Misbah Uddin

1.0 Objective

When traversing an enclosed area, there are different methods to move towards a certain goal. This final project highlighted the use of Open-CV, or using CV2 in python for ROS, one such method that allows a robot to overlay and filter out colors in a camera to act as “sight” for a bot. By experimenting with past methods of movement and scanning, it is possible to create an efficient method to traverse an enclosed area, using different colors as goals. In this case, that goal is to identify four different colored walls in sequence: orange, white, yellow, and black, with a simulated robot in Gazebo recognizing, approaching, and taking a picture of each while avoiding any obstacles along the way. The created program in this report will be referred to as `Final.py`

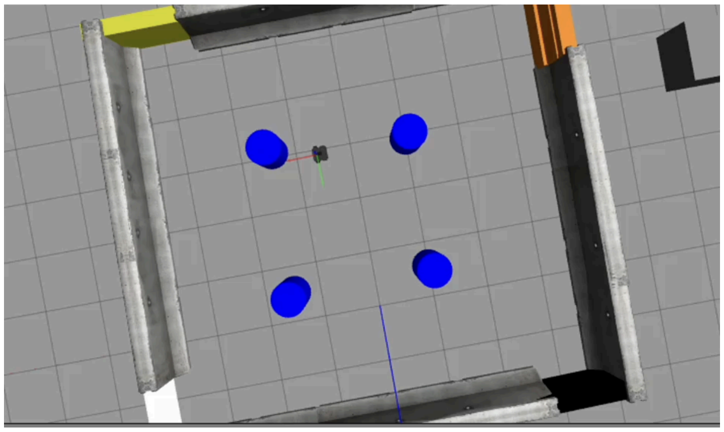
Section 2.0 will describe the program used for this task to be completed, in particular the 2 parts it consisted of. Section 3.0 will explain in detail about the first node, color tracking, and Section 4.0 about the second, obstacle avoidance. Section 5.0 will mention trial and error, and how certain portions in the final assignment code were fine tuned for efficiency. Section 6.0 will bring up potential improvements and conclude the report.

2.0 System Design

The Final.py code makes use of 3 topics and 2 nodes. `MotionTopic`, `ImageTopic`, and `laserTopic` are all crucial to both the movement and detection capabilities of the simulated T3 bot, each topic referring to a separate scanner that exists on the robot to send and receive data. ImageTopic for visual cameras, showing what the robot “sees”, MotionTopic to manipulate movement based on acceleration or velocity, and laserTopic to receive data from the attached LIDAR device.

Due to `MotionTopic` being used to manipulate the robot’s movement, both nodes in `Final.py` are subscribed to `MotionTopic`. However in detection, a node for color tracking makes use of ImageTopic in order to know where the robot should move, while Obstacle avoidance uses laserTopic to sense the bot’s proximity to objects.

Figure 1: The enclosure that the robot was simulated in, the final task being to avoid the 4 pillars in the middle to get from Orange, White, Yellow, and Black in that order.



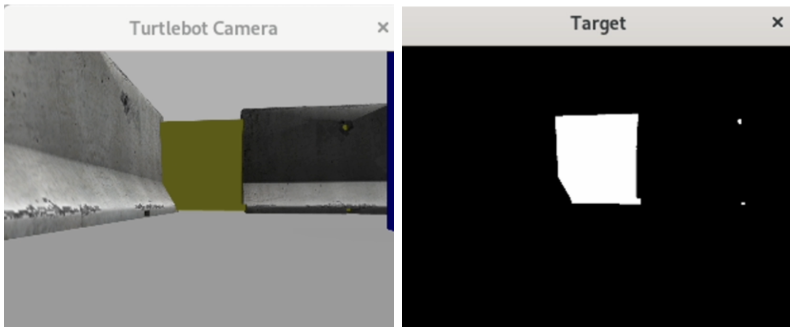
To recognize each color, the Color Tracking node uses `ImageTopic`, creating 2 camera views. One is a simple first person perspective of the robot as it travels through the enclosure, and the other fully in black except for the desired color goal, overlayed by white to make it stand out and signify to the robot this space is important and therefore a goal to traverse to.

In between diagonals, for example Orange to White or Yellow to Black, there is a chance that the robot can crash into the pillars serving as obstacles. Since it is commanded to move towards the space of white pixels, the pillar isn't visually seen. In this case, the second Obstacle Avoidance node would subscribe to laserTopic to detect nearby obstacles and adjust movements to prevent collision.

3.0 Color Tracking Node

The color tracking node itself has 2 parts to consider, scanning and movement. In the Final.py file, this node is named `trackNode(targetCols)`. TargetCols is an array of each aforementioned color, each index in the array having its own 3 parts, the color name, the lower range, and the upper range. These ranges are Red-Green-Blue values that the computer uses to recognize certain colors. The node itself runs through the TargetCols array, filtering out any colors outside of the aforementioned color range in order to consider it as a goal.

Figure 2: The unfiltered vs filtered Camera Views

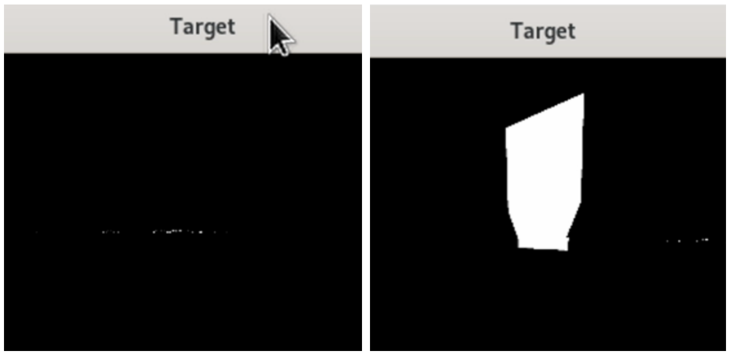


Unfiltered Filtered

Once the upper and lower bounds of the desired color goal's range has been inputted, the robot will spin in an estimated 360 degree circle to see if there is a clear path to the wall that matches that color. If not, the bot will wander aimlessly and then try again. Both of these make use of a `CycleCounter` Variable, acting as a makeshift timer. The `CycleCounter` starts at 0, and counts up the 200 as the robot spins at a rate of 0.4 m/s. Once breaching that, `CycleCounter` is set to a negative number, counting from -80 up to 200 again, with the time it takes to traverse from -80 to 0 allowing for random wandering and repositioning.

Both of these processes are interrupted once the robot locks onto a significantly dense area of filtered pixels, over 1 million. Due to the nature of a simulated environment, there are spots on the enclosure that are not the wall, known as pixel noise(See Figure 3 for an example of noise vs an expected target). These small spots may confuse the robot, making it see this spot as its goal and traversing to it when it should not.

Figure 3 - Pixel Noise vs Target



Noise Expected

Once locking onto these denser areas, the robot takes into account the height and width of the walls, in order to find values for the variables `delx` and `dely`. These two variables represent distance between the robot and the filtered out color area, `delx` being the central point of the x-axis of these areas, and `dely` doing the same for the y-axis.
The variable `delx` is used to calculate the angle, twisting the robot left and right so that its robot is always facing the center. For `dely`, 60% of the height down of these areas is used to find a distance from the bot, with a general rule that if it would get further away, the distance would get lower, potentially negative. As the bot is already centered by angle to face the wall, this section only involves moving the robot forward, letting the Obstacle avoidance node help avoid any potential collisions. Thus, the robot is expected to approach the goal until it falls between a range of 310 to 340 to finally take a picture and move on to the next goal.

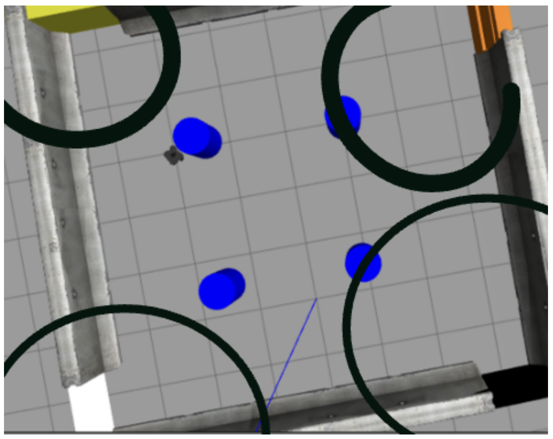
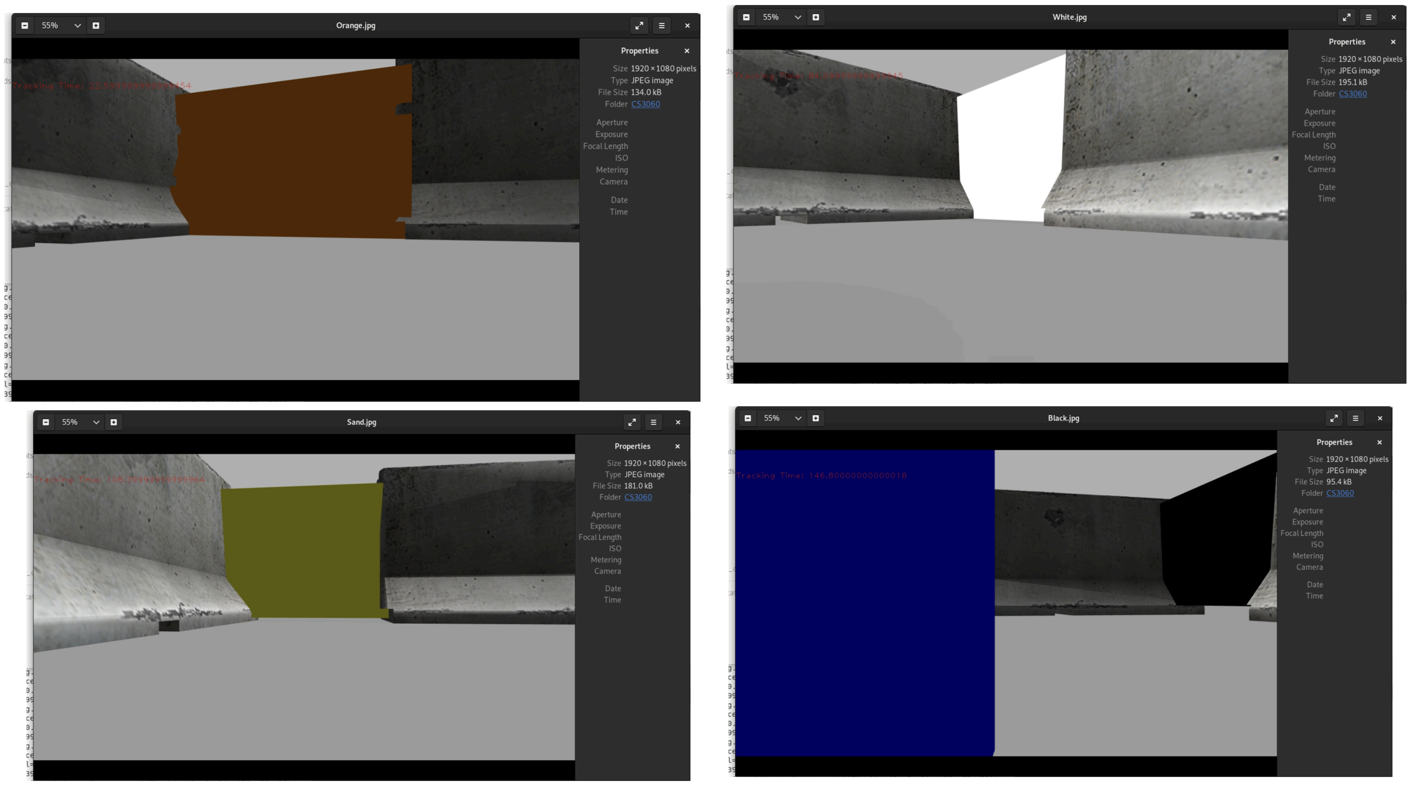


Figure 4: A general area where the robot pause to take a screenshot, considering its goal as "found"

Figure 5 below shows the final screenshots taken from using trackNode(targetCols):



4.0 Obstacle Avoidance Node

Unlike the color tracking node that subscribes to `ImageTopic`, the obstacle avoidance node, which is `callback_laser(msg)`, subscribes to `laserTopic`. This topic makes use of a bumper system, taking into account laser range data 30 degrees left and right of the robot, and setting a left or right boolean value as false until an object is deemed as too close.
There are 3 possibilities that can occur, either an obstacle on the left side of the bot is too close, activating the left bumper, the same occurs but on the right side, or both bumpers are marked as true indicating an obstacle is directly in front of the robot.
Should an object be on the left or right of the bot, instead of stopping, the robot will instead move at half of its original speed, and slightly turn to the opposite direction so that it would continue moving, with the assumption that once the obstacle is not longer possibly able to collide, then `delx` would be used to center the robot back to its original course towards its goal. This allows the robot to continue moving, still getting closer to its goal instead of stopping. Both bumpers being activated is the only scenario where the bot would reverse, scanning again for its target to continue its task.

5.0 Testing

Making use of `dely`, adjusting a filter for pixel noise, and the adjusting the time for spin and wander scanning were the most involved aspects of `Final.py`
Given that `dely` is the level height from the top of the wall downwards to the bot, finding a suitable distance was difficult in a simulated environment. In order to test a suitable range, the simulated robot was made to run another file known as centerTarget.py, which already incorporated `delx` to center the orientation of the robot towards the center of a filtered pixel area regardless of distance. Since `dely` varied based on distance, this file was used to test acceptable values, since the robot also did not move, only rotate when centerTarget.py ran, meaning there would be little room for error. This process involved much guesswork, manually moving the robot to various points in the area adjacent to the orange wall, and printing out the value of `dely` in the center console. Since the simulation provided live camera feed, it was possible to tell when a distance was too close or too far creating a roughly acceptance range for `dely`.
Pixel noise was similar in this regard, since originally the program would act once the area of pixels was greater than 0, so any filtered out pixels would be singled out and made a goal. This also meant that a singular and insignificant pixel that falls on the filtered range, despite not being on the colored walls but on the enclosure wall, would “trick” the robot into seeing it as a goal. To adjust for this, values greater than 0 had to be tested out, starting with smaller numbers like 50, and slowly working potential area values to 500, 1000, 3000, to finally the acceptable 1,000,000 pixel dense areas being acceptable to break the spin command and approach.

The spin and wander timer is another implementation that required similar testing to adjusting for pixel noise. If the robot was ever in a scenario where it was behind a pillar and was just far enough it didn't need to back up, then the bot would spin endlessly with no possible way to ever reach its goal, so a wander feature was implemented as a backup. Various numbers for `SpinCycle`, which is the upper limit of `CycleCounter` before it must be reset, were tested in order to ensure that the robot had to have made a 360 degree turn before implementing wander. On top of this, wander also needed to run for a limited amount of time, since even a small adjustment would be more effective if the robot spun to look for its goal afterwards, rather than wandering endlessly without the same 360 degree search. To ensure such a small adjustment and assuming that the rate `CycleCounter` updated by 1 was consistent, a smaller negative value was used to placate wander’s timer as it counted up to 0, so a large adjustment from the robot’s original position is minimized.

6.0 Summary and Conclusion

The final system that was created after all these adjustments was a program that would take a simulated Turtlebot3 in a gazebo environment, spin that robot until it sees its end goal's target color filtered out for a significant area, and rush the bot towards it, adjusting its angle to the center of the goal until falling between a distance of 310 to 340 from the wall. In the case of not seeing this goal, `Final.py` would instead let the bot wander for a short period of time before trying again. Should it approach an obstacle, its speed would slow down and then the robot would slightly adjust its angle to continue its movement before adjusting back.

Each small adjustment made was done through extreme trial and error, as each run of `Final.py` brought to light more edge case scenarios. The first of these were figuring out what distance the robot would approach the wall up to before moving on. While we did use estimated values of `dely` in order to find an approximate area, it is possible that based distance based on pixel density might have been more efficient. Using approximate ranges for `dely`, while doable wasn't the most accurate, and even ended up with one of the goals being screenshotted behind a pillar since it did technically fall within range. This pixel density implementation wasn't figured out mostly due to frustration from the slow speed of the Azure VM for testing.
Testing speeds for the robot would also be another potential improvement, since a lower speed was used by default from previous tests, where the robot would crash headfirst into a wall or pillar and the obstacle avoidance node was not updated fast enough to prevent this. In turn, testing the ropsy.rate value is another potential improvement to lessen the possibility of this.

Overall, the project demonstrated significant progress in using visual sensing with real-time obstacle avoidance. This project successfully met its objectives and provided valuable insights into navigation and obstacle avoidance in simulated environments based on color as well as laser scanned distance.