Final Project Writeup: ISEC DeliveryBot

CSC 325: Robotics

Dani Massa

Tyler Nass

Purpose

The purpose of our final project was to design and implement a delivery robot. We decided to have the robot's location for exploration and navigation be within the Integrated Science and Technology Complex (ISEC) building at Union College. Our goals for this project were to successfully map two rooms within ISEC, develop python scripts that allowed for multiple delivery destinations, and incorporate user input on a remote PC in order to select these destinations. Additionally, we attempted to include a facial detection framework, which was used to ensure that a human was present at a given delivery location to "pick up the package" (no actual physical packages were delivered due to the TurtleBot's small structure). Although we were successful in implementing the three previously listed goals, we were unable to fully integrate the facial detection framework.

Accomplishments

For this project, we were able to fully map two rooms. Additionally, we succeeded in developing a python script that allowed for the robot to traverse to multiple delivery destinations. These destinations included four locations in one of the rooms, and a singular location in the second room, for a total of five destinations. We ensured that our python script handled user input to select these locations. Finally, we modified our python code such that the robot was given a start position; each time the python script is run, the TurtleBot moves to its start position prior to embarking on its delivery journey.

Struggles & Process: Mapping

We started our project by first determining which areas in ISEC would be the most successful to generate a map for. In regards to what makes a location good for mapping, we avoided any areas that contained glass windows and walls that stretched all the way down to the floor. Additionally, we tried to stray away from locations that contained stairs and structural gaps such as railings at the height of the Turthebot's sensor. We were cognizant of the fact that these types of locations may result in inaccurate measurements of the room. Moreover, there was a high probability that these types of locations would generate an overall inadequate map that would be unusable for the TurtleBot to navigate through. These constraints existed because the TurtleBot uses a lidar sensor for its mapping techniques. The lidar sensor uses individual laser beams to build a map, meaning that anything made of glass is technically invisible to the robot. Additionally, the TurtleBot collects data on these laser scans within the two dimensional plane that scanner is situated on. This means that any obstacles that do not intersect this plane, such as tabletops, chair seats, etc., will not be detected by the TurtleBot. Instead, these types of objects are typically mapped as small dots, which represent the legs of the table/chair.

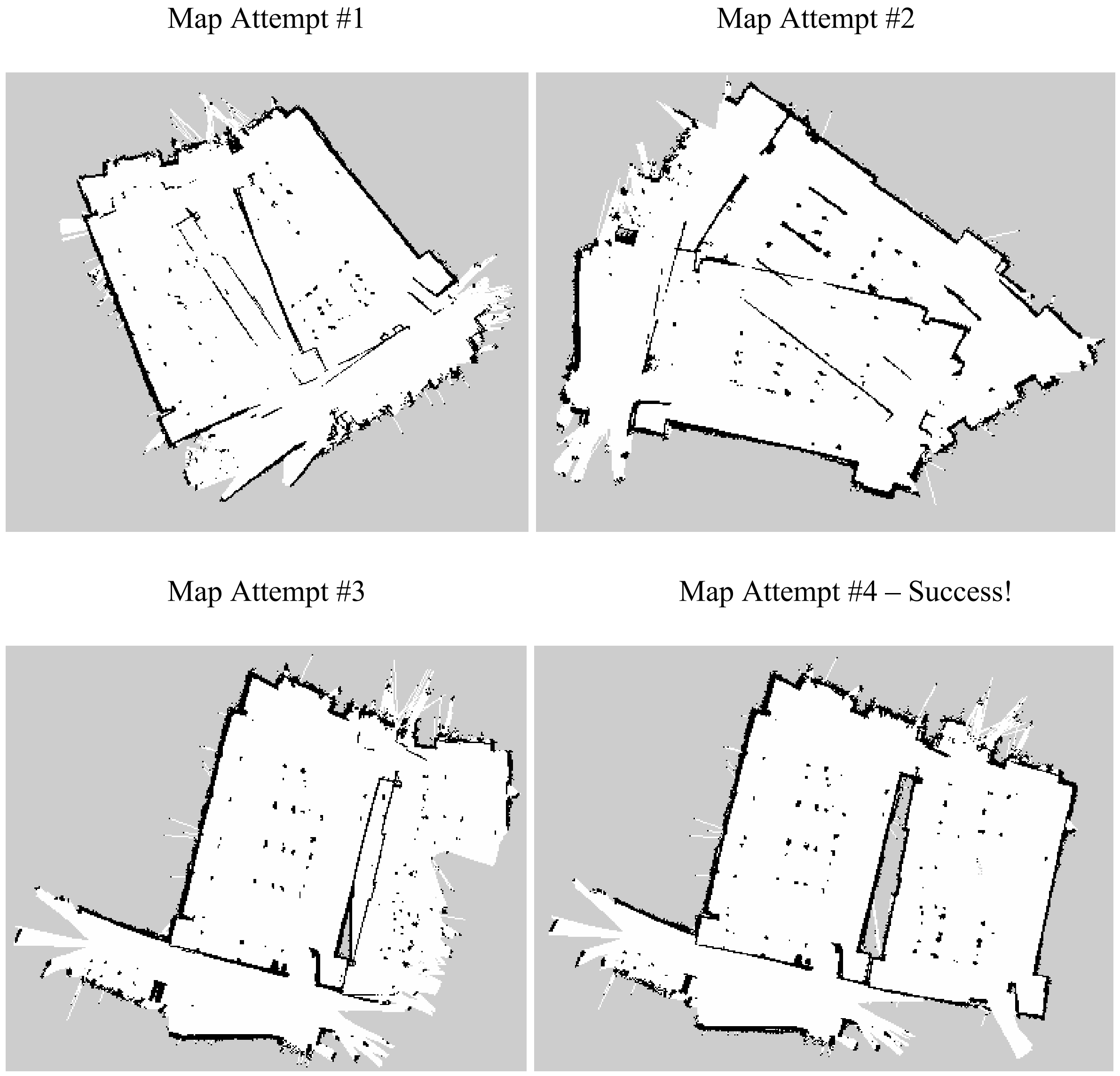
The two rooms we selected for our final project are ISEC 178 and 180, the ECBE Advanced Teaching Labs. However, prior to mapping these locations, we decided to do a practice map in the Bailey building on campus. We did this mainly to get comfortable navigating the robot in an environment that is far away from the remote PC, and to figure out effective forms of communication during the mapping process. We decided that the best approach mapping would be for one group member to control the TurtleBot on the remote PC and have the TurtleBot's camera active for live video footage, while the other group member traveled with the TurtleBot and gave additional feedback and directions via telephone. After completing our practice map, we began to map the two rooms in ISEC.

We quickly realized that one of the biggest challenges we would have to overcome was preventing the TurtleBot from generating a map in which the two rooms overlapped. ISEC 178 (Room 1) and ISEC 180 (Room 2) are located right next to each other, and they share a wall.

Although the robot was able to handle mapping out one room quite well, once we navigated it to the second room, the map had the tendency to shift and become inaccurate. Overall, there was an added complexity from mapping one room versus two rooms; the TurtleBot had difficulty lining up the shared wall between the two rooms. Additionally, the robot needed to explore a relatively large area, so it frequently became confused as to where its current position was on the map.

Moreover, another major challenge we had to overcome was that the TurtleBot's battery life is very limited. During several of our attempts to map the rooms, the battery would die prior to completion. Nonetheless, slow movement and frequent rotations are imperative to a successful map. Therefore, there was a tradeoff between fast movement to prevent battery life, and risk of error. In total, we had four attempted maps (shown in Figure 1), in which the first three contained overlapping rooms, and the fourth did not. ISEC 178 (Room 1) is located on the left side of the map, ISEC 180 (Room 2) is located on the right side of the map, and the hallway is located at the bottom of the map. See next page for Figure 1.

Figure 1: Progression of Map Development Within a Turtlebot Framework



We solved the issue of overlapping rooms by carefully altering the way in which we navigated the TurtleBot through the environment. To be more specific, we concluded that one of the best ways to avoid errors in mapping was to retrace our steps. By having the TurtleBot move through areas it had previously mapped out, we were able to fix any overlap or errors the robot ran into. For example, our fourth map originally contained overlap similar to maps one through three. This overlap occurred once we shifted from exploring the first room to exploring the second room. However, once we fully navigated through the second room, we traversed back into the first room. We paid close attention to the shared wall, and slowly moved the TurtleBot parallel to it. By doing this, the robot was able to dynamically fix the map, and as shown in Figure 1. This notion of retracing our steps was particularly important when handling unexplored areas of the map as well. We discovered that by having the TurtleBot traverse through areas it already mapped prior to exploring a new area, we were able to reduce the amount of errors within the overall map.

Our final map was only partially completed; we were unable to fully explore the hallway located at the bottom right of the map due to the restrictions on battery life. The designated starting position of our robot was located in the explored area of the hallway (the bottom left of the map). The unexplored hallway area created an issue because the robot's built-in pathfinding algorithm determined that the fastest path from the robot's starting position to Room 2 was through the unexplored hallway. Since this area was not fully mapped, the robot's pathfinding algorithms got confused, and as soon as the Trutlebot entered unmapped space, it would start moving slowly and inconsistently. Our solution to this problem was to move the TurtleBot to Room 2 only after it had moved into Room 1 (i.e. Room 2 can only be our second destination, not our first). In an ideal world, the solution to this issue would have been to create a better map with a robot that contained a larger battery lifespan, or to figure out a way to merge two maps together so to form one fully completed map with no gaps. However, since the map that we created took four multi-hour attempts to finalize, and because we were unable to get a map merging library to function, we decided to implement the current map we had already created as best as possible.

Struggles & Process: Python Script & Facial Detection

The first challenge we solved while writing our python script was to ensure that the user was only able to enter valid inputs for the number of delivery destinations for the robot to travel to. This input needs to be an integer between a given lower and upper bound inclusive. To do this, we implemented a try/except block, in which we try to cast the input to an integer. If this throws an exception (i.e. the code fails to cast the user input to an integer) the program prints an error message and quits. Then, using an if statement, we check whether the input is between the upper and lower bounds. We originally intended the upper and lower bounds to be modifiable, but our current implementation of the code does not support visiting more than two locations. This is because the section of the code that sets the coordinates for the destinations is hardcoded to set either one or two sets of coordinates. This hardcoded issue could be fixed by implementing a loop that generates a list of tuples containing coordinates. Additionally we would need to refactor the code to contain a new function that takes the list of tupled coordinates and iterates through them, rather than taking a set amount of individual variables that depends on the number user inputted delivery destinations.

After determining the number of destinations, we ensured that the program outputs a list of all possible destinations, and prompts the user to type one of them in. Once the user has submitted their input, a for each loop checks the list of destinations to find a match. When a match is found, the robot's x and y instance variables are assigned to the coordinate values associated with that destination. If the option for two destinations was selected, the program will prompt the user to input a second destination. When it outputs a list of possible destinations, the destination that was selected for the first location will not be displayed. However, it is still possible to type in the name of the first selected destination (this is a pitfall/possible error). Once a second destination is selected, the robot's x2 and y2 instance variables are assigned to the coordinate value of the second destination. If the number of destinations is one, x2 and y2 will remain at their original initialized values of 0. These instance variables are never accessed in the case of only one destination being selected. Once the coordinates for the destination(s) have been found and loaded into the appropriate variables, the position_movement() method is called on x and y. If there was a second destination selected, the script will also call position_movement() on x2 and y2.

It should be noted that we added a hard coded start location for the robot. Thus, when the python file is called, the robot will first navigate to the start location, prior to prompting the user for input destinations. This models a delivery depot, where the robot would pick up the requested deliveries in order to transport them. This also allows for a more controlled setup; once the robot was in the start position, any delivery would begin from the same known location.

After reaching a destination, the robot waits a couple of seconds before traversing to any subsequent destinations. Our original plan was to implement a facial detection feature at this point, such that the code would use the TurtleBot camera and OpenCV to search for faces within its field of view. Moreover, the robot would only search to find a face, it would not try to recognize the specific face. When the robot detected a face, it would take a picture with its camera and save that image. This implementation was meant to model confirmation delivery messages used in real world package delivery systems (ensuring that a person is present to pick up the package). Once the photograph containing a face had been saved, that delivery would be considered complete, and the robot would be free to move on to its next delivery location. Once the robot completes its final delivery, we hoped that the program would terminate successfully.

Unfortunately, during our attempt to implement this facial detection framework, we ran into major bug issues. Although we understood the logic behind putting this code into effect, we struggled with syntax and python-specific errors. One of these challenges included automating actions we would manually trigger through command line calls in the terminal. We tried to implement Python's os module in order to execute commands (specifically the command that tells the TurtleBot to take a picture via its camera). However, the os module only allowed us to execute commands on the remote PC, not the raspberry pi on board the TurtleBot. In order to circumvent this issue, we needed to redesign the layout of our code. Rather than attempting to automate images being sent from the TurtleBot to the remote PC, we tried to incorporate the OpenCV library instead.

In regards to utilizing OpenCV, we were cognizant that this framework would already be used for facial detection. Therefore, it made more sense to use OpenCV on the video feed from the TurtleBot's camera, and save a singular frame from that video, instead of taking a picture on the TurtleBot and sending it to the computer. Nevertheless, we were ultimately unsuccessful in getting OpenCV to function within our program. This is because we were unable to get the program to connect to the TurtleBot's camera. We encountered the following error:

```
faces = face_cascade.detectMultiScale(self.image, 1.1, 4)
cv2.error: OpenCV(4.5.3) /tmp/pip-req-build-afu9cjzs/opencv/modules/objdetect/src/
cascadedetect.cpp:1389: error: (-215:Assertion failed) scaleFactor > 1 &&
_image.depth() == CV_8U in function 'detectMultiScale'
```

We were unable to resolve this error, which is ultimately why we were unable to implement the face detection portion of our plan for this project. Since the facial recognition did not work, we needed to find another way to model the pick-up part of the delivery process, so we had the robot wait for four seconds after arriving at the first destination before moving on to the second destination.

Conclusion

Despite issues surrounding the implementation of facial recognition and OpenCV, we were still able to create a fully functional delivery robot for our final project. Additionally, we gained extensive experience in mapping, and we were able to overcome a variety of challenges surrounding the creation of adequate maps for TurtleBot navigation. Moreover, we were able to compile previous material from a variety of labs within this class, in order to produce a singular project demonstrating what we have learned within the past ten weeks.