Ryan Masson
Lab Partner: Cary Murray
EECS 301: Intro to Robotics
Professor Argall
11/14/16

Assignment #2: Navigation— Localization, Planning, and Mapping

ABSTRACT

In this lab we programmed our robot to be able to represent its world as a centralized, metric map. We implemented a wavefront propagation algorithm that enabled the robot to plan and execute a path from a given starting point to a given ending point, if it had knowledge about the world. We also implemented the Backtracking Spiral Algorithm, a complete coverage algorithm for mapping a grid maze. On the first demo day, our planning algorithm did well and we won the competition; however, on the second demo day, our BSA implementation was not able to show its power because of the inaccuracy of our localization. The robot kept crashing into walls.

I. INTRODUCTION

Up to this point, the abilities of our robot have been rudimentary, when thought about from a general notion of intelligence. We started with actuation and sensing, which is just being able to move and read infrared signals. We moved on to feedback and reactive control that kept the robot away from walls and other obstacles. But we had not yet implemented mechanisms that begin to simulate the higher-order, intelligent behaviors of animals. In this lab, we researched and programmed localization, planning, and mapping, as an introduction to the higher-level capabilities of robots.

Localization is the ability for a robot to know its location in the world. To do this a robot needs to have a specific kind of representation of location. These representations are often called *world models*, and *maps* are the most commonly used type of world model.[1] There are many forms for a robot's map to take, including odometric maps, landmark-based maps, topological maps, and metric maps.[2] These different forms have their own advantages in different situations. A robotics researcher must consider things like whether the world will change, how large the world is (for computational complexity), and which objects or tasks actually need representing. Once a robot has a reliable way to represent the world for its specific world, it can strive for more advanced functions.

Planning is the ability for a robot to use what it knows about the world to generate and follow a path from its initial state to some goal state. This involves *searching* the robot's world model. Searching is never a simple task; there is no one search algorithm, because "sometimes searching the complete representation is necessary … while at other times only a partial search is enough, to reach the first found solution."[3] Planning is thus a challenge for the robotics researcher. She or he must consider time complexity, space complexity, the extent of the available information, and how the plan will actually perform in practice.[4] However, these challenges are worth the struggle, because enabling planning is a vital step towards making advanced autonomous robots.

The next step is mapping. A robot with mapping capability can build the world map by actively exploring its environment, and then plan on that built representation. Sometimes the robot has limited information about the map and explores it; sometimes it has no information about the map and has to

---

[1] Matarić, p. 146
[2] Matarić, p. 146
[3] Matarić, p. 152
[4] Matarić, p. 155-157

build as it goes. In the latter situation, if the robot also does not know its location and must represent it somehow, it may use SLAM (simultaneous localization and mapping). SLAM is a big challenge, because the robot must do the processes of localization and mapping in parallel. SLAM also presents the *data association problem,* which is "the problem of uniquely associating the sensed data with absolute ground truth."[5] Luckily, in this lab, we had a map of known size and starting position, and so did not need to combat the harder challenges in robot mapping.

II. METHODS

Localization

The first task in this lab was to enable our robot to move around the world and represent its current location. In our context, the world was an 8 x 8 square grid, with the top left corner as the origin, (i, j) = (0, 0). From the origin, *i* increased in the "down" or "south" direction, and *j* increased in the "right" or "east" direction. Our robot had to be able to stay oriented while traversing the grid, keeping itself in the center of the traversed cells and maintaining a heading in one of the four cardinal directions.

First, we decided the way to represent the current location in the robot. To us there appeared one option that could be the correct approach: a centralized map representation, where the robot's current position was stored as an (i, j) tuple. In our situation, a distributed map as part of a behavior-based system seemed like it would add needless complexity. We did not need the flexibility and higher level of abstraction that a behavior-based system can provide.[6] Storing the location as a tuple was a simple way to do the computation necessary for localization.

We wrote a function called *move* that took three arguments: the number of cells to move, the current heading, and the current position tuple. The function moved the robot the specified number of cells in its current direction, updated the position depending upon the heading, and returned that position. To move the correct distances, we used simple timing. We timed how long it took the robot, at our chosen motor speed, to move from the center of one cell to another, and then passed a *sleep* command to the ROS node with the observed time. This represented moving a distance of one grid cell. We also wrote functions called *leftTurn* and *rightTurn* that turned the robot 90 degrees to the left or right.

To test the localization capability of our robot, we had the robot move from one position to another, and print its final position, with three different pairs of starting points and ending points: (0, 0) to (3, 0), (0, 0) to (0, 3), and (0, 0) to (2,2). At this stage in the lab, we did this by writing a sequence of calls to the *move* and *turn* functions by hand. The tests would show that the robot could accurately move in the grid space and that it could internally represent its position.

Planning

Next, our robot had to be able to generate and execute a plan that moved the robot on a path from any given starting position to any given ending position in the grid. As we knew from lecture and the textbook, "finding the optimal path requires searching all possible paths, because if the planner does not check all possibilities, it may miss the very best one … therefore, this is a computationally complex process."[7] There are then several ways to address this complexity and make planning tractable, such as computing local paths or implementing heuristics that choose any path before the best path. But with our grid cell environment, a discrete and small environment, we did not have to worry that much about the computational tractability of our approach. We implemented a simple wavefront propagation algorithm that chose the lowest cost path from a starting position to a goal position.

First, we enabled the top level of our system to have access to four command line arguments: starting position, starting heading, ending position, and ending heading.

---

[5] Matarić, p. 229

[6] Matarić, p. 189

[7] Matarić, p. 228

Then we wrote a function called *fillCostMap*, which took the current map object and a goal position. The function used the functionality of the map class to assign costs to grid cells. We first set the cost of the goal cell to zero and initialized the wavefront to be a list of the unblocked neighbors of the goal cell. We then wrote a loop that said "while the wavefront is not empty, increment the current cost, set the costs of the cells currently in the wavefront to the current cost, add their unblocked neighbors to the wavefront, remove the old cells in the wavefront, and repeat." This modified the map object so that every grid cell ended up with a cost corresponding to the distance from the goal position, which was equivalent to the number of moves it would take for the robot to reach the goal from the start cell.

The next major function was *genPath,* which took the updated map object and a starting position as arguments. This function iteratively generated a path from the starting position to the goal, represented as a list of tuples, by successively choosing the cells with the lowest cost. The first tuple in the list was the starting position. Then, the function looked at the neighbors, chose the one with the lowest cost, added it to the list, and repeated on the new cell it just chose. When finished, it returned the list of positions that represented the path.

Finally, we used a function called *genComms*, which took the path list, the starting position, the starting heading, and the goal heading. This function, depending on the current heading and position, iteratively generated and executed a series of calls to the *move* function to move the robot on the path generated by *genPath*.

To test this planning capability, we decided to pass three sets of start and goal positions to the system, and see how quickly and accurately the robot could move from one point to another. This would show that our robot really had the ability to move itself according to an internal representation of location in the world.

Mapping

The final task was for our robot to build a map of its world through exploration, and then be able to plan paths through the built representation. We researched and evaluated some different approaches before settling on our approach, the Backtracking Spiral Algorithm. The approaches we considered were:

- Wandering: This would be a random exploration of the grid. This was the first consideration, but we wanted to look at more systematic methods that could assure a complete exploration of the grid space.
- Depth-first search: We read online that one can treat a grid maze as a graph and perform a depth-first search to traverse it.[8] However, we realized that this would result in lengthy backtracks, which would waste a lot of time and slow our performance.
- Latitude/Orientation method: In the book *Planning Algorithms,* we read about an algorithm for maze searching that did not require the robot to store its current position in world coordinates, and instead maintained the latitude —number of cells in the north direction from the initial cell— and the orientation, whether the current loop path was clockwise or counterclockwise.[9] We found this to have potential, but we couldn't quite figure out how to implement it.
- Backtracking Spiral Algorithm (BSA): We found this in a paper published online in the IEEE journal. It was a coverage algorithm that ensured completeness and good speed with a clever backtracking mechanism. We chose to implement it.

Our implementation of BSA was essentially divided into two modules: a spiral module and backtracking module. The spiral module operated under the following logic:

[8] "Graph Traversals for Maze Search."
[9] LaValle

IF (obstacles_all_around)
THEN ending_spiral_point_detected

IF (NOT obstacle_in_left)
THEN turn_to(left) and move_forward

IF (obstacle_in_front)
THEN turn_to(right)

OTHERWISE move_forward [10]

Key to this approach were the obstacles: obstacles were both actual walls, detected by our robot's sensors, and "virtual obstacles," which were the cells already visited by the robot. With these obstacles and the above logic, the robot entered a spiral pattern, setting the real obstacles around it in its map as it went along. When it reached the end of a spiral, the code invoked the backtracking module.

The backtracking module backtracked to the closest unblocked point to which the robot could have gone in the execution of the spiral. We stored these potential backtracking points in a stack data structure, so that we could pop off the most recently added position when invoking the backtracking module. With this position in hand, the code set a path to the point, executed it, and resumed a new spiral. If implemented correctly, we hoped the BSA would provide a complete and fairly efficient exploration of the grid space.

To test our mapping algorithm, we planned to run it from each of the four corners of the grid maze and see how far it would get, how quickly it could go, and how many times it would collide with the walls of the maze. This would test the coverage, speed, and accuracy of the exploratory mapping.

Best Practices and Improvements

In this section I will describe two Best Practices for robot programming, and explain how we used them to retest two of our behaviors in this lab.

Throughout this course, we have relied on a lot of specifically tuned timing to move our robot in the world. For both moving and turning, we have let the motors spin, timed how long it takes for the robot to reach a desired location or position, and plugged that time into a sleep timer in the Python code that runs our ROS node. If we were giving advice to someone who had never programmed a robot before, we would advise two things about timing. First, we would say to avoid it. But if necessary, one should tune the timers with speed and accuracy in mind. In our testing of localization, we wanted more speed and accuracy in heading, so we sped up the motors and spent more time tuning to see if we could get better performance out of using timer statements.

Another general principle that we have come to is the necessity of thinking about hybrid approaches to a robot problem, where you combine reactive or feedback modes of control with deliberative modes of control. In our initial implementation of the BSA, we spent so much time on the correctness of this deliberate, top-down algorithm, that we neglected a wall following mechanism that would keep our robot on course while walking the maze. As we were depending on timing for tight moves and turns, we could not respond to the accumulation of small errors that eventually sent the robot crashing into walls, ending

[10] Gonzalez et al., p. 2041

the exploration. To improve this area, we added a wall following mechanism to the mapping code. We retested the mapping performance to show that the hybrid approach would work.

III. RESULTS

Localization

We placed the robot in the middle of a tile representing the origin, and executed a sequence of moves that we wrote by hand. We used a stopwatch on a phone to measure time. We measured the error in distance from the center of the cell with a yardstick, measuring both in the east-west direction and the north-south direction. We placed a protractor (printed on a piece of paper) over the robot at the end to measure the error in its orientation.

Start to end	Distance error (inches)	Orientation error (degrees)	Time (seconds)
(0, 0)S → (3, 0) S	0.50 E, 0.50 N	5º left of S	15.99
(0, 0)S → (0, 3) S	1.25 W, 1.00 N	S (perfect)	17.04
(0, 0)S → (2, 2) E	1.00 W, 1.50 N	2º right of E	20.57

Planning

We placed the robot in the starting position and heading and started the planning algorithm. We measured time, distance error, and orientation error in the same way as before. We ran each path until it successfully reached the goal. Here I record a coordinate pair if the robot got stuck and a distance error if it successfully reached its goal cell.

Path (start to goal)	Run number	Final position / Distance error (inches)	Orientation error (degrees)	Time (seconds)
(2, 1)S → (6, 2) E	1	(4, 1)	Stuck on E wall	50.43
	2	0.50 E, 2.00 N	7º left of E	1:04.19
(7, 6)W → (4, 7) E	1	1.13 E, 2.75 S	5º right of E	44.94
(7, 2)N → (0, 3) W	1	(1, 0)	Stuck on W wall	46.21
	2	(3, 0)	Stuck on E wall	38.47
	3	1.25 W, 1.50 N	13º left of W	1:03.23

On demo day, our planning mechanism did quite well. Since the paths were fairly short, there was not enough time for errors in heading to accumulate and cause much crashing. We won the competition against robots that had wall following mechanisms because wall following tended to waste some time. But this discrepancy would come back to bite us on the next demo day, where we tested our mapping.

Mapping

On the mapping demo day, our robot covered nine cells at the most (out of the 64) and kept crashing into walls. This happened because of the accumulation of errors in the heading after turning. However, when we did our own testing after the competition, we had slightly better results, including a huge run that covered 39 cells. In these tests, we placed the robot in the center of one of the corners of the maze, headed "south" (we called south the direction the robot faced if it had a wall to its right as it sat in a corner). We recorded time with a stopwatch app, and watched to count the number of wall hits and number of cells covered. We did one run for each of the four corners.

Time elapsed	Wall hits	Cells covered
1:06.95	3	11
3:33.94	1	39
1:08.54	1	10
0:39.00	1	5

Improvements

We redid testing for localization and mapping, after implementing the improvements discussed in the methods section.

Localization:

Start to end	Distance error (inches)	Orientation error (degrees)	Time (seconds)
(0, 0)S → (3, 0) S	3.25 E, 0.50 N	7º left of S	12.05

Mapping:

Time elapsed	Wall hits	Cells covered
2:56.97	3	22
1:08.00	2	13
1:46.66	2	19
1:54.03	3	20

IV. DISCUSSION

As mentioned before, our dependence on timing for accurate moving and turning worked for simple localization and planning, and we won the competition on the first of the two demo days. However, as soon as our robot had to do more extensive walks through the maze during the mapping demo day, it broke down. If we had to do this again, we would spend much more time implementing a wall following mechanism that would allow the BSA to show its true power in the realm of 8 x 8 grid maze mapping.

To go back further, we were at first uncertain about whether we should have kept the full battle droid platform from the previous lab or dismember it and put our robot on wheels. We decided on the latter, and in retrospect it was the correct choice. If we had stuck with the battle droid platform, our performance would have been much worse, because of its slowness and inaccuracy in walking. This showed us a process of really thinking about the problem at hand and adapting, instead of trying to force something we had done before to fit the new problem.

Our first Best Practice, to improve the speed and accuracy of localization, backfired. In the new test, we had a much bigger distance error and orientation error. However, we did lower the speed it took to reach the cell by about four seconds. If we used this code with our initial mapping algorithm that did not have wall following, it would have made our performance even worse, because the errors would accumulate faster. However, an adequate wall following algorithm could deal with it, and benefit from the better speed.

After implementing a basic wall following algorithm, our mapping performance improved. We had slightly more wall hits, but actually more cells covered. This shows that a hybrid approach with higher-order thinking augmented with lower-level feedback control can protect a robot from fatal errors, making it more robust in the real world.

V. CONCLUSION

This lab was a productive introduction to localization, planning, and mapping. We enabled our robot to know its position in the world and move from one place to another, whether it knew about its world at the start or did not. With these solid fundamentals as a base, a robotics researcher can move on to ever more advanced capabilities.

REFERENCES CITED

"Bioloid Premium Robot Kit." *Trossen Robotics*. N.p., n.d. Web. 03 Oct. 2016.

Gonzalez, E., O. Alvarez, Y. Diaz, C. Parra, and C. Bustacara. "BSA: A Complete Coverage Algorithm." *Proceedings of the 2005 IEEE International Conference on Robotics and Automation* (n.d.): n. pag. Web.

"Graph Traversals for Maze Search." *CSE 326*. University of Washington Computer Science, n.d. Web. 14 Nov. 2016.

LaValle, Steven Michael. "Algorithms for Maze Searching." *Planning Algorithms*. Cambridge: Cambridge UP, 2006. N. pag. Print.

Matarić, Maja J. *The Robotics Primer*. Cambridge, MA: MIT, 2007. Print.