Maze Exploration Using Custom SLAM Implementation With Odometer and Variable Number of Distance Sensors

Burgunder, M., Ener A. E.
Università della Svizzera italiana

***Abstract*—Using a self-resizing map Python implementation, we present a novel method of performing Simultaneous Localization and Mapping (SLAM) for a robot. We present the algorithm by incorporating the Python source code into a ROS2 package, and simulate a Robomaster robot in the Robotics simulation software Coppelia.**

***Index Terms*—SLAM, Robomaster, Coppelia, Bresenhem's Line Algorithm**

I. Introduction

Solving mazes can take a number very well documented and researched methods, all of which solve a given maze in finite time. One of the simplest methods is by the "right/left-hand rule", whereby an explorer follows the wall on either side. Note, that this method only words if walls of the maze are all connected, which is relatively common. Maze solving is a particularly interesting and important problem to solve for autonomous robots, as robots are becoming increasingly prevalent in our roads and pathways.

While maze solving has been known since ancient times[2], creating a physical mechanism for mapping the layout of the maze with a high degree of accuracy is not. Once a map is created, one can implement path-finding, another class of algorithmic problems, with which one can guide a robot to a given destination. Several algorithms exist with which one can implement path-finding , such as modifications of Breadth and Depth First Search, or Dijkstras algorithm.

In order to create the map however, there need to be sensors that feed data into the map creating mechanism. For this, a collection of software modules known collectively as Robot Operating System (ROS) has been created to interface a robot with computational algorithms and path finding techniques for a wide range of different environments. For these techniques to work, it is necessary to estimate the location of the robot before mapping can be done, which togehter is known as Standardized Localization and Lapping (SLAM) algorithms.

For this paper, we the wall-following technique to explore the maze, while implementing a custom mapping the maze for further processing. In particular, we develop from memory management techniques scalable mapping of the environment to large size with a custom specified degrees of precision.

Finally, we simulate the given algorithm in Coppelia, a robotics simulation software which interfaces with ROS. It is here, that we simulate a maze, and are able to demonstrate the effectiveness of our implementation.

II. Sensors

To simultaneously map the environment and move the robot, we have, in addition to the ROS specified controller, implemented a class in Python 3, that maps the environment. To do so, we have implemented 2 basic metric types that are collected continuously as the robot drives around, via the publish/subscribe model in ROS. The first is the *Odometer*, which is a sensor that estimates the location of the robot. The other type of sensors we use are *Distance* sensors, which use lasers and light detection to estimate the distance to an obstruction.

A. Odometer

The odometer is a basic sensor that tracks wheel movements on in order to estimate to what position the robot has moved to in space. Given that we are running our technology on simulation software, the odometer will in general be very accurate. For our choice of robot, the Odometer will be tracking the movements of all four wheels.

B. Distance Sensors

We have implemented 2 sensors located on either side of our robot which gives us 4 directions in total for data collection. For our purposes, we decided to face 1 sensor forward, 2 of them right and the remaining to the left. This way we can detect walls on 3 sides and move accordingly. Having 2 sensors on the right side allows us to go parallel to the walls that we are following and reduce noise in our mapping.

For each sensor, we input the position of the sensor in relation to the position of the robot, as well as the angle in which the sensor is pointing in. This way, we can determine the position and direction of the sensor by combining the pose given from the odometer, and the change in angle/position in the sensor.

C. Navigation

For navigating the maze, we are implementing the right-hand-rule, whereby the robot follows the wall that is on their right hand side. With the help of the right-pointing distance sensors, it maintains a constant distance to the wall. When it drifts away too much, it can detect and move towards the wall, and if we gets too close, it moves away. For a given threshold of error, if the sensor readings are identical, the robot goes straight until they are not.

Since the robot always has a wall to its right only, we will only ever need a left turn (except for initially moving towards the wall). Combining this with the simple maze structure, which only has 90 degree angles between walls we can safely say, that if the forward facing sensor detects a wall, we need a 90 degree left turn to continue our exploration. Since nothing is perfect, even in simulations, we support our turns with the distance sensors to realign ourselves with the wall.

Given these constraints, we have created a basic maze in Coppelia with no disconnected components, where we are able to navigate the maze to find the exit.

III. SLAM

A. Map Architecture

Given that a robot should explore its environment without restrictions to the environments size, it requires a mapping algorithm that can dynamically scale, at runtime, to accommodate new points for which no memory has been allocated yet. A straightforward approach creates a large 2D matrix with integer values. From there, one can transform the initial estimated position of the robot into one that corresponds to the matrix coordinates.

The problem with this approach however, is that the robot needs to be able to explore a larger and larger space. For points that are of high value in the columns, this is trivial: one simply appends an integer into the columns. Similarly, for a point that has been detected further above one can simply add a new row.

Where this data structure fails however, is when points are detected that are below, or to the left in the map. In order to accommodate these points, a new row needs to be inserted into the 0th position in the matrix. Similarly, for new columns, a new 0th column needs to be inserted into each row. This is problematic for several reasons.

First, in order to place items in the beginning of rows or lists in general, all the other data in that list needs to be shifted by 1 position. Using a simple benchmark, found in Appendix A, the process of unshifting rows or columns is slower by a factor of 10^7, at around 0.1. For SLAM, this is very inefficient, given that the time of flight (ToF) sensors generally release data points faster than this.

Secondly, the modified position needs to be recomputed. While not necessarily difficult, it becomes tricky to be able to say how the inputted data relates to the data structure in question. This way, any third party analysis of the data is difficult to achieve successfully. Furthermore, if the computer running ROS were to somehow crash, unless modification data is saved continually (along with the actual map), a restart of localization cannot occur.

For this reason, we have decided to create 4 matrices that simulate a singular matrix, which form the large matrix on which the robot can map its environment on, without the need for additional saved data. Logically, each point is treated with positive coordinates that are the matrix coordinates in a particular square, i.e. (smaller) matrix. Depending on the negativity/positivity of the incoming sensor data, each point is assigned to a corresponding square, whose coordinates are also determined by this function. This way, for any new point for which the saved matrix is too small, the matrix can be extended by appending new rows or columns.

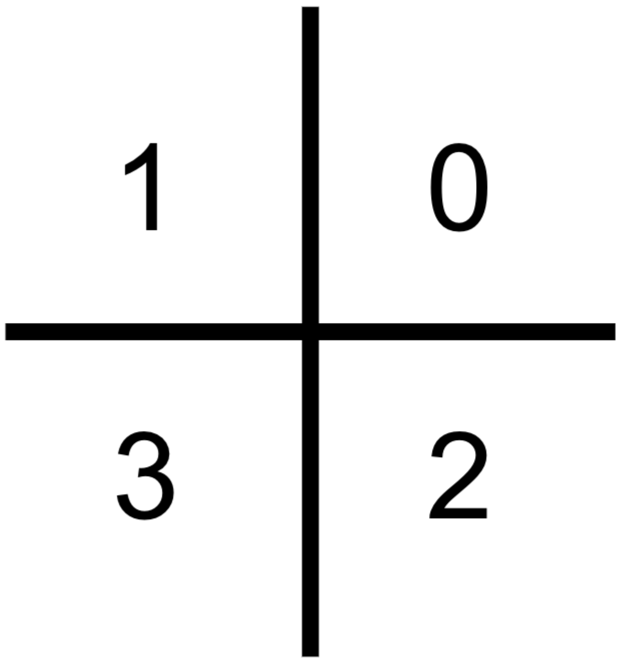

Fig. 1. The numbers represents the name of the squares, which have a secondary use as sums, subtrahends, etc. to gain valuable information on where the Bresenhem line fill will occur

Like this, one can transform any new point coordinates from ROS space, to the logical system and vice-versa.

B. Line of Sight

Given a point that one of the ToF sensors has detected, we know that there is an obstruction at this point. From this follows, that anywhere in between, is empty space, which we record in the map.

Transforming both the pose of the sensor (which is usually very similar to the position of the robot) and the position where an obstruction has been detected into word coordinates, we need to determine which matrix positions we need to fill out with ”empty”. To do so, we use Bresenhem’s Algorithms which does this to a relatively high degree of accuracy.

Given the 4 square architecture, we have decided to compute the line of sight pixels for each square separately. A better approach would have simply been to compute the coordinates in world coordinates, then transform these into our 4 square architecture. However, we have implemented the former way, due to the iterative approach of creating the software, hence why we have fallen into the recency bias trap. Nonetheless, our implementation works just as well as the more elegant approach.

C. Output

For output, we have at first, simply printed the map matrix in the CLI. Once some of the smaller problems were solved

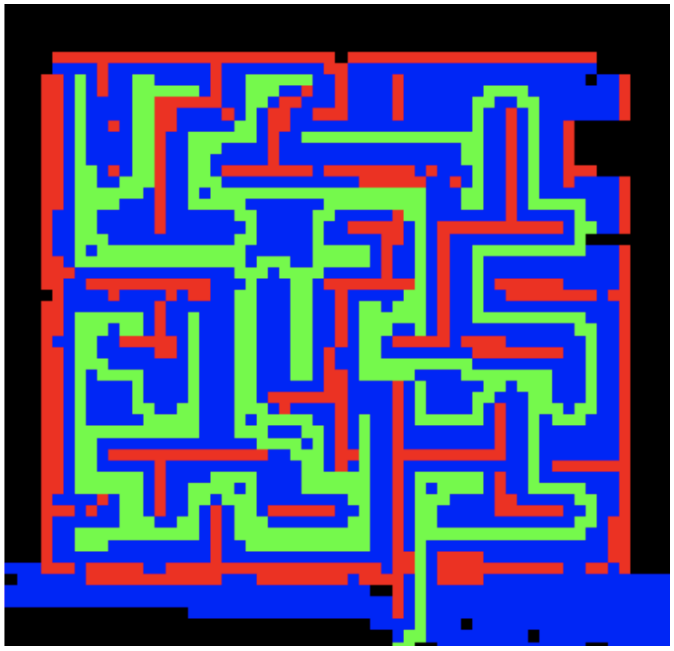
Fig. 2. Given the robot starting up, in the middle of the image, the basic maze appears from sensor data using SLAM. Note, that the origin point of the logical map (where all four squares intersect) is not necessarily in the middle of the image

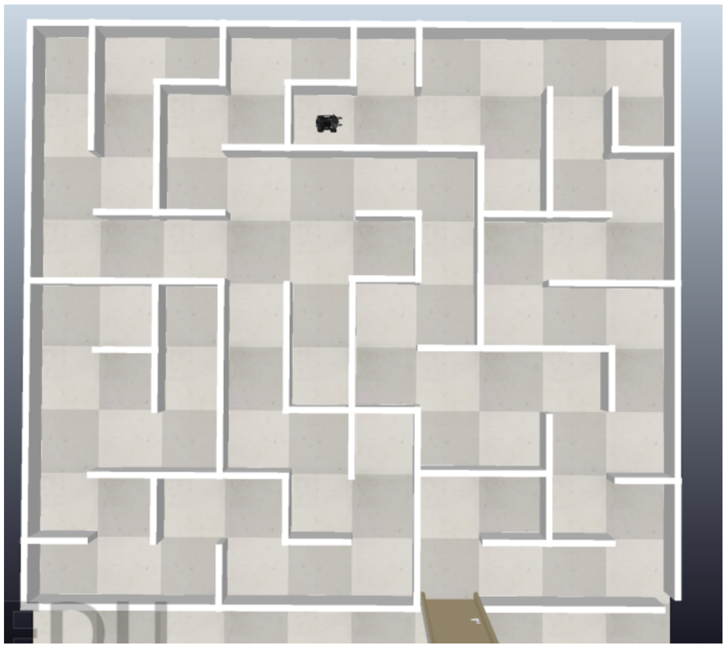
Fig. 3. The design of the maze in Coppelia. Comparing this to Image 3, one can see that the SLAM algorithm maps the existing maze in the simulation. There are slight errors in the mapping, such as missing walls, which can be attributed to low resolution.

and matrices larger than around 15 elements, we decided to save the matrix as a color map with 4 colors:

1) Black: Undetected/unexplored points
2) Blue: Empty space
3) Red: Obstruction
4) Green: Either a current, or past pose of the robot

D. Experimental Setup

For the setup, we have placed the sensors on the Robomaster as stated further above. We have also created a simple maze which is laid out in a grid. To simulate the narrative of waking-up-somewhere-without-recollection-of-how-you-got-there (i.e. ”mornings drunkards realization”), we’ve placed the robot in a corner of the maze, without any points included in its memory. The maze itself has exactly one exit out which is located on the other side of the maze, to provide ample opportunity for SLAM to prove itself.

IV. Results

The initial results of the SLAM algorithm look promising. Given in Figure 3, we can see that the robot has explored the entire maze, and found the exit at the bottom of the image. Note that this is a low resolution map. With higher resolution, the walls are not as perfectly straight, and errors creep into the simulation, as can be seen in 4. With too high resolution, we may also obtain non-sensical maps, which suggests that the resolution of the mapping has an effect on the robot, likely due to inefficient use of processing power. The overhead for map computation should be use for pose updates and sensor data processing. Foregoing these mechanisms, causes the callbacks to be delayed, eventually causing the robot to navigate the maze in unexpected and incorrect ways.

V. Conclusion

As we see, implementing a custom SLAM algorithm, while a challenge, is not entirely without its benefits. Once the initial architecture is in place, the base code is infinitely customizable.

VI. Further Work

The current work acts as a prototype for far more many features, optimizations and architectural changes. In addition to the implementations already mentioned, localization of the robot currently depends on the Odometer, which assumes that the robot in the simulation doesn’t crash into any obstruction. Instead, the Iterative Closest Point Algorithm (ICP) can be used modify the odometer reading to reflect crashes and possible friction if the robot were to realized outside of a simulation.

There is also considerable mystery as to the best approach on which points to be overwritten, and which ones not. We have not allowed for overwriting of obstruction points, which means that the mapping only works with a static environment.

Path planning, such as those done with the RTT algorithm, could also still be implemented. Alternatively, we can use a custom algorithm, as described in [1], or other path planning algorithms.

Regardless of these upgrades, the most significant change

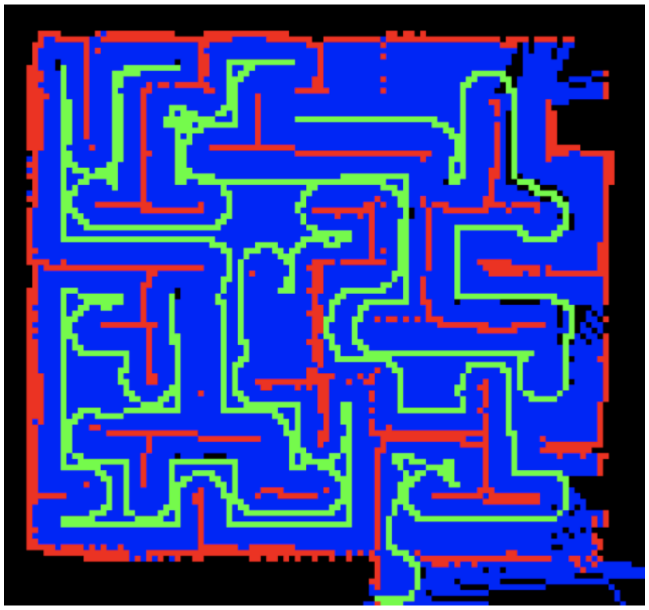

Fig. 4. Inputting a higher resolution for mapping causes the positions of recorded points to become more exact. However, errors emerging from determining which matrix point to update, and implicit errors from the sensors, lead to fuzzy walls, in the final mapping.

would be to change the architecture in such a way, that the node controller runs on a separate system than the SLAM algorithm. With the navigation detached from the processing of the sensor data, we can increase the resolution without compromising navigation. Another solution would be to simply save data points, and create the map at a later point.

The recording of each point in the map can also be improved, by allowing each recorded point to save variational readings, and in this way, dynamically modify the color of that point. This way, one can get probabilities of what may be located in any one point.

Given this probability space then, the line of sight algorithm can be improved by using a more modern line drawing algorithm such as the Digital Differential Analyzer or Wu's line algorithm.

Acknowledgments

Each author would like to thank the other author for their contribution to the paper. Special thanks to those colleagues who contributed to the project indirectly by ways of companionship.

References

[1] https://www.youtube.com/watch?v=rop0W4QDOUIt=1s, 'Maze solving', 2017. [Online]. Available: 02.06.2022

[2] https://www.rental-center-crete.com/blog/labyrinth-of-the-minotaur/, 'Everything about the Mythical Labyrinth of the Minotaur (Minoan Maze)', 2020. [Online]. Available: 02.06.2022

Appendix

To measure the running time of inserting a row over placing a row at the beginning of the matrix, we have benchmarked this by the below code. We have attempted to follow best benchmarking practices for measuring the interested in functions.

```
import time
import numpy as np

one_row = None
final = []
for i in range(1000):
    row = []
    for j in range(1000):
        row.append(j)

    if one_row == None:
        one_row = row
    final.append(row)

def measure(one_row, iterations, m=False):
    s = 0
    f = 0
    for i in range(iterations):
        if m :
            s = time.time()

        final.append(one_row)
        if m :
            f = time.time()
            print("append operation:")
            print(f-s)
        if m:
            s = time.time()

        np.insert(final, 0, 0)
        if m:
            f = time.time()
            print("unshift")
            print(f-s)
    return f - s

print("start measure (1000 elements)")
measure(one_row, 100)
res = measure(one_row, 1, True)
```