ME 169 Final Report: Implementing Continuous Space Mapping

Tyler Nguyen, Lorenzo Shaikewitz

I. Introduction

In ME 133b we simulated a compelling alternative to grid-based mapping that emphasized continuous space. The basic idea was simple: instead of dividing the world into grid points that are either ”occupied”, ”free”, or ”unknown”, assume the entire world is free space and store the exact position of obstacles as they are encountered. To prevent discretization, obstacles were stored as line segments. Our results showed clear memory advantages over a grid without any major issues for high-level motion planning, but were limited by the simulation's assumption of perfect odometry and perfect sensing.

With this project, we implement continuous space mapping on an ME 169 robot equipped with LiDAR (RPLIDAR A1). For simplicity, we retain the assumption of perfect odometry and focus on the simpler problem of pure mapping. Our specific goals were to use continuous space to map obstacles under the following assumptions:

- Our sensor is not perfect, and may sometimes detect obstacles in places where there are none.
- Obstacles may move, disappear, appear, or change shape.
- The robot must not hit any obstacles during its mapping routine.
- Sensor integration only occurs while the robot is stationary.

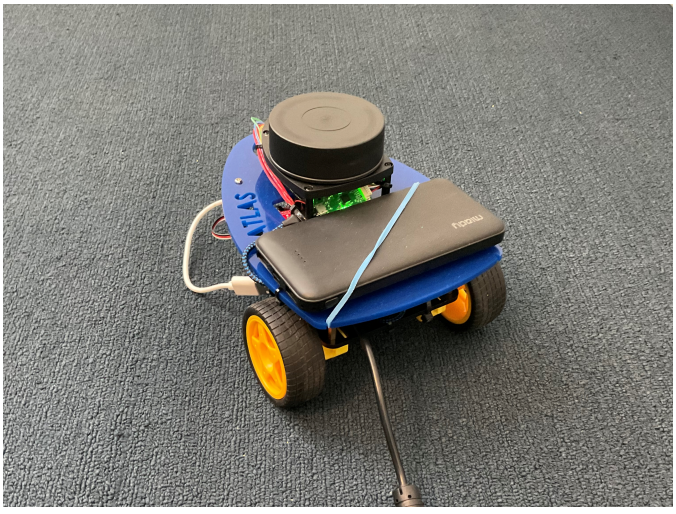

Fig. 1. A close-up view of the robot used. The robot has two motorized wheels and a ball roller. The LiDAR is mounted on top of the robot.

Although we had implemented this concept in simulation, the addition of sensor noise required a significant reformulation of the problem. The result, though not complete, shows the strong potential of using line segments for mapping and offers key lessons for future improvement.

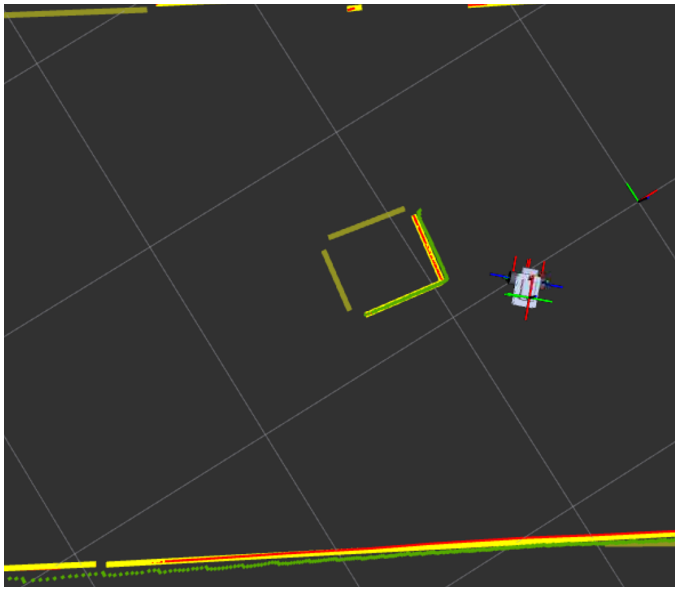

Fig. 2. A screen capture of the mapping algorithm in action! The dark yellow walls are saved walls that are no longer directly visible, while the bright yellow walls are the visible ones. In a hallway with a box, the robot successfully detected the box after circumnavigating it.

II. Algorithm

The basic mapping algorithm converts incoming sensor data into lines and merges this data into an existing line map. Specifically, it:

A) Converts incoming sensor data (points) into Cartesian coordinates and then into lines.
B) Preprocesses the existing wall list, associating each sensor line with a nearby wall and removing existing walls that the latest scan sees through.
C) Updates each wall according to the new sensor data.
D) Post-processes the wall map, eliding walls by slope and adjacency.

The resulting wall map was used for local planning, which could be extended to a higher-level motion planning algorithm. The robot's initial position and orientation define the origin and axes of the map. Also note that the process of associating

lines with nearby walls may be used for minor corrections to odometry (localization), but this feature was not implemented.

A. Sensor Data Conversion

The first step in continuous mapping is converting the discrete sensor data into lines. The RPLIDAR node handles reading the sensor and transmits a LaserScan message. We use the LaserProjection library to convert this LaserScan into Cartesian coordinates. This library automatically compensates for the slight delay between laser points and any effects from turning. The resulting PointCloud2 message contains a list of points in the robot's base frame.

These points are received in a separate node and converted into lines using a series of least squares regressions. Assuming the list of points is ordered by LiDAR angle, we begin with the first point and continue adding points to the regression until the least squares error exceeds some threshold. Then, the longest line that falls within the threshold is saved and the process repeats with the next set of lines. To summarize:

1) Begin with the first two points from the LiDAR scan (ordered by angle).
2) Add a point to the list.
3) Compute the least squares regression line of the points in the current list.
4) If the error is less than a threshold, save the regression line and repeat 2 and 3. Otherwise:
5) Add the previously saved regression line to the list of scan lines. If there is none, add the line between the first two points in the list.
6) Select the next two points and repeat from step 2 until there are no more points.

This algorithm proved quite effective at accurately converting points to lines (see Figure 4. Its main parameter, the error threshold, must be carefully tuned to achieve good performance. This threshold balances information loss from the conversion to continuous space: a low threshold fails to filter out minor sensor noise, while a high threshold hides small features. Although sensor noise increased with the distance of an obstacle, we used a single threshold primarily tuned for nearby walls. Another source of error was from angle wrapping. Without proper wrapping, the line is always split into two at the branch cut.

B. Wall Preprocessing

After converting the latest sensor data into lines, the algorithm fuses this update into its existing wall map. On a high level, this fusing combines sensor data with nearby previously measured walls, then checks the whether the remaining previously measured walls are visible with the current sensor sweep. If the walls are not visible, they remain in the map. If the walls should be visible (but are not near any of the sensor data), they are immediately removed. Practically, this means we average new obstacle data with the existing map but assume the sensor never sees through a true wall.

The specifics of this set of algorithms are tuned to produce output useful for wall updating and post processing. First, the preprocessing code pairs every sensor line with a previously measured wall line, if possible. The paired wall line is determined from distance and slope similarity thresholds. If multiple walls meet these criteria, the wall with a midpoint closest to the scan line's midpoint is chosen. This approach is simple but not optimal: it sometimes selects walls that are disjoint but near the scan line's endpoint, and it can fail if the optimal wall is very long. However, in most cases it generates an accurate wall-to-scan mapping.

Before handing the remaining walls, preprocessing re-examines each paired wall for partial occlusion. Any walls that are partially occluded by measurements they are not associated with are split into associated portions and occluded portions. We measure partial occlusion by converting each line's endpoint into polar coordinates and finding the angles where scan lines are in front of wall lines.

Lastly, we process each of the remaining walls into two categories: blocked and visible. Since we are only interested in full blockage, we use a line connecting the midpoint of the wall to the robot's current position. If this line does not intersect any scan lines it is marked "visible" and removed from the map. Conversely, if a line intersects a scan line it is marked "blocked" and remains in the map. Note that this does not properly handle a partially blocked obstacle that disappeared or shrunk.

C. Wall Updating

With the existing walls converted into blocked, visible, and paired walls, the next step of our mapping routine is to update the locations and slopes of the paired walls. This algorithm is as follows:

1) Select a scan line (order of scan lines does not matter).
2) If the scan line has no previously measured wall associated with it, add the scan line to the map. Otherwise:
   a) Let $\hat{v}$ be a unit vector in the direction of the scan line and $\hat{w}$ be a unit vector in the direction of the associated wall such that $\hat{v} \cdot \hat{w} > 0$.
   b) Let $\hat{a}$ be unit vector in the direction of some predefined linear combination of $\hat{v}$ and $\hat{w}$ (weighting $\hat{w}$ slightly more).
   c) Compute the normal to $\hat{a}$ pointing from the scan line to the wall line as: $\hat{n} = (\hat{v} \times \hat{w}) \times \hat{a}$.
   d) Compute the projection of the distance between the scan line's endpoints and the wall's endpoints onto $\hat{n}$. That is, if the scan line has endpoints with vectors p_1 and p_2 and the wall has q_1 and q_2, compute $d_1 = \mathrm{pr}_{\hat{n}}(p_1 - q_1)$ and $d_2 = \mathrm{pr}_{\hat{n}}(p_2 - q_2)$.
   e) Define a new line with endpoints $r_1 = p_1 + d_1$ and $r_2 = p_2 + d_2$. Add this line to the map.
3) Repeat from step 1 until all scan lines are processed.

This procedure produces a series of walls that are around the same length as the measurement lines but have an averaged slope and position. Long previous wall lines are broken up into smaller lines, allowing recognition of gaps (from for example, an opening doorway). This process heavily weights

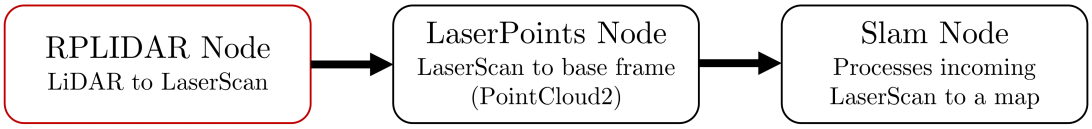

Fig. 3. An overview of the basic structure of the code. The RPLIDAR node is provided and converts LiDAR data into a LaserScan message. The LaserPoints node converts this message into the base frame, compensating for timing. Lastly, the Slam node performs mapping, integrating information from odometry. A separate structure of nodes handles odometry collection, local planning, and low-level control.

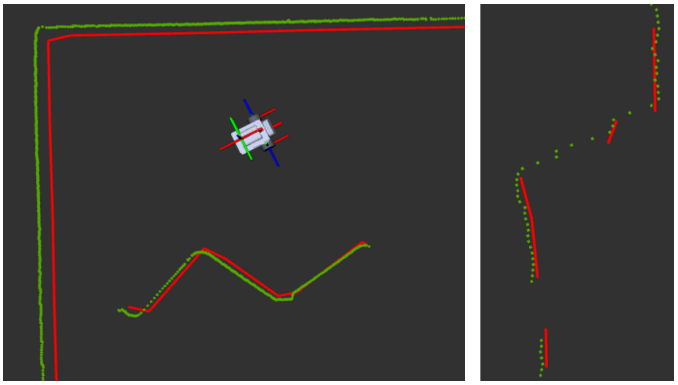

Fig. 4. The line detection algorithm at work. Light green points are the raw LiDAR laserscan message, while the solid red lines are the instaneous fitted lines. Near the robot (left), the line fitting does a good job approximating the sensor data. Further from the robot (right), the error threshold is too low to fit long segments of data.

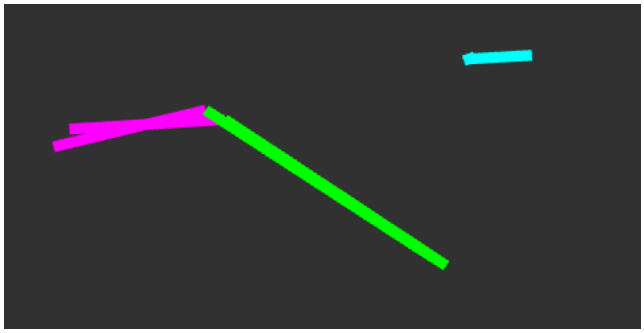

Fig. 5. A simple example of pairing sensor data with pre-existing map data. Given three wall lines and three lines from the latest sensor data, the pairing algorithm pairs the two closest lines, allowing for some distance and angle variation. In this example, lines of the same color are paired.

new measurements, enabling quick recognition of a changing environment but high sensitivity to noise. Without proper care, the procedure can also lead to improper shortening of existing walls that are partially occluded.

D. Post-Processing

The final step in the mapping algorithm is to post-process the map, connecting nearby lines and pruning unnecessary ones. This process uses a list of all line endpoints to 1) merge any endpoints that are reasonably close together and 2) merge adjacent lines with similar slopes. Notably, this method does not remove overlapping walls.

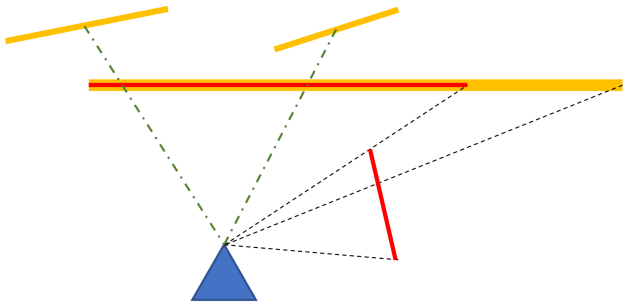

Fig. 6. The robot's occlusion checker operates differently for unpaired and paired walls. The unpaired walls (shown above) are simply checked using a line from their midpoint to the robot's origin. If the line crosses a sensor line, the unpaired wall remains in the map. Otherwise, it is removed. This is vulnerable to missing lines that may have been shortened, such as the line to the left. For paired lines (right), the robot uses an angle sweep to identify and keep the segment that is partially occluded.

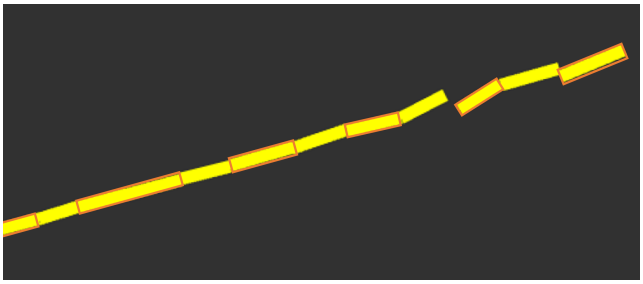

Fig. 7. Without post-processing, long walls are broken up into a series of short walls. Every other wall is highlighted in orange to show this effect.

III. Evaluation and Limitations

A. Qualitative Assessment

This project focused on using lines in continuous space to recognize obstacles and create a map. We begin with several qualitative assessments of the quality of the produced map. Figure 8 shows two different maps generated by the algorithm. Without motion, the system reliably produces a fairly accurate and stable map. Especially as the lines get further from the robot, however, the map becomes increasing unstable and inaccurate. This is a result of the uniform offset applied to all sensor ranges; using a variable offset would ensure increased line stability.

With motion, the map is clearly quite incomplete. Hidden lines are discontinuous, despite once harboring a continuous line. This effect traces back to our algorithm for merging lines: existing walls are automatically shortened to the length

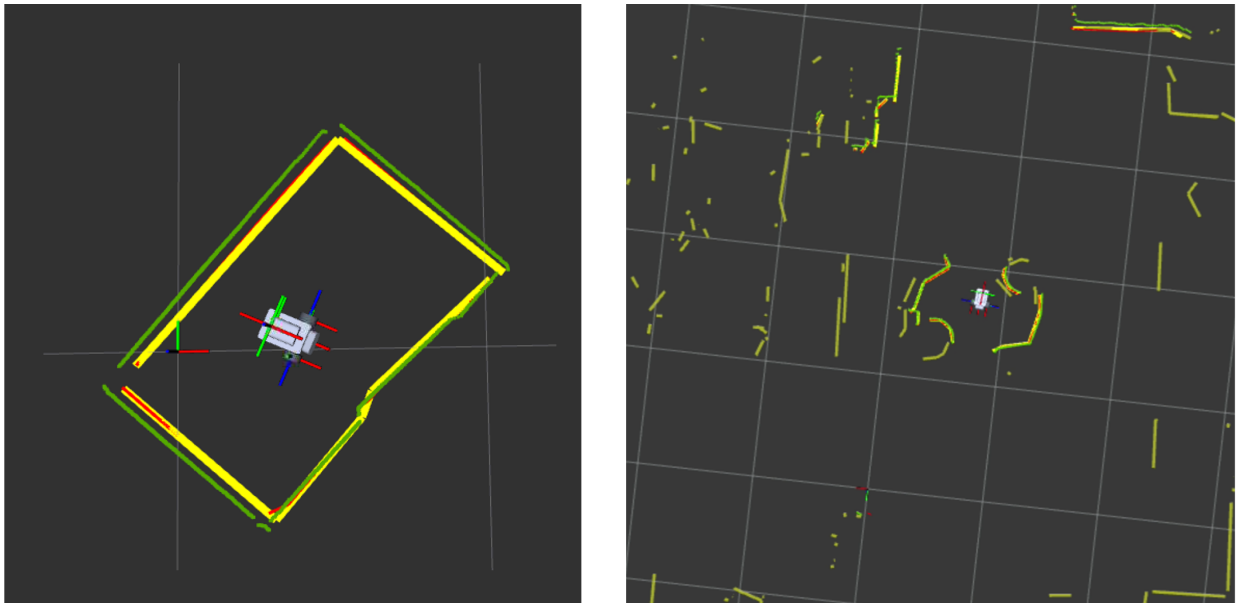

Fig. 8. Two different maps generated by the robot. In a simple map (left), the robot can see all walls and identify their location, although it misses parts of the corners. In the more complex map, it still accurately identifies visible obstacles but has only a rough map of previously explored areas.

of the latest scan lines. As the robot turns a corner, the soon-to-be-occluded wall becomes shorter and shorter. Our solution, estimating where walls were partially occluded, was not reliable enough to truly fix this issue.

Additionally, we implemented a local planner that checks if a path intersects or comes close to any lines in the map. This was the most successful result, with every local plan tested exhibiting correct behavior. Qualitatively, these continuous space maps are useful for planning in visible space, but only usable as a rough guide for planning in previously explored space.

B. Quantitative Results

We also evaluated the mapping algorithm for speed and memory in various environments. Though this algorithm is quite memory efficient, using continuous space requires significantly more computational resources. It is worth noting that the low number of wall lines allowed us to use brute force for nearly every algorithm; the current version of the code leaves opportunity for significant optimization. Speed trials were conducted using a Raspberry Pi 4 computer.

Three trials were conducted in two different spaces: one simple space, small with no obstacles and right angles, and one more complex space. Each trial was recorded after some robot exploration of the space. Table I shows the average memory requirement (number of lines, where each line is four integers) and run time per scan. Note that scans were acquired at 6 Hz and processed much slower. Additionally, Table I shows a breakdown of the run times for each method of the algorithm described above.

These results show some important trends with our code. In a simple map, processing routines are quite fast and the slowest process is sensor conversion. In contrast, the more complex map has a faster sensor conversion run time but a much slower preprocessing routine. Sensor conversion is slower in the smaller map because lines are longer; longer lines contribute to larger matrix multiplications to compute the least squares error. As expected, preprocessing is much slower with a larger map. This traces back to the brute force algorithms used to compute nearby walls; with more than 200 walls in the map, these algorithms take significant time. Map updating, a simple process, is very quick in both trials.

TABLE I
LINE COUNTS AND RUN TIME FOR A SINGLE UPDATE FROM AN INCOMING SCAN, BROKEN DOWN BY SUBPROCESS.

	Simple Map	Complex Map
Line Count	9.33	202
Total Runtime	560.2	1341.6
Sensor Conversion Runtime	510.6	211.6
Preprocessing Runtime	13.0	945.6
Map Updating Runtime	29.3	65.3
Post-Processing Runtime	5.9	118.8

IV. Further Possibilities

Perhaps the most exciting result of this project is that continuous mapping is possible because it enables a host of further possibilities. While generating a map in continuous space is difficult with discrete sensor data, lines make high-level planning much more interesting. This starts with local planning, which is as simple as checking whether a specified path crosses one of the given walls. In fact, it may be beneficial to perform simultaneous localization and mapping in grid space and convert the grid into lines for motion planning, although this conversion introduces its own set of challenges.

Another extension of this project could use the continuous mapping to identify macro-scale objects. Additionally, our

continuous-space map is ripe for an implementation of Tangent Bug, or an exploration method, or generation of a Voronoi diagram. Here we consider the latter possibility in some detail.

A. Voronoi Diagram From Continuous Space

The use of lines to identify obstacles could enable an efficient dynamic generalized Voronoi diagram implementation. Suppose the robot has perfect (or at least reliable) localization and mapping that outputs the robot's position in an incomplete map with obstacles.

To compute the local Voronoi diagram, first pair each line with the two closest lines, recording the position where the closest line changes. Assuming no lines can intersect, the closest line can only change if the new closest line has an endpoint. This simplifies finding the exact position of closest lines: only line distance from endpoints must be computed. After the two closest lines are identified, compute the known Voronoi diagram. Using this local Voronoi diagram we could implement some form of Tanget Bug, moving towards the target until an obstacle is reached, then following the Voronoi diagram around the obstacle until the path to the target is no longer blocked or the obstacle is completely circumnavigated. Like in tangent bug, in the latter case the algorithm could move to the closest reachable meet point to the goal (making sure not to select the same meet point from a given obstacle twice) and repeat.

Alternatively, wall lines could be used to perform a gradient decent until a point equidistant from the two closest lines was reached. Then, the robot could remain on the Voronoi diagram until it had a clear path to its destination.

V. Discussion

There were some key disappointments and many surprises during this project. First, dealing with sensor noise proved much more difficult than anticipated. On a grid, incoming sensor measurements can just be used to update their respective grid points, perhaps with some confidence or Gaussian distribution. In continuous space, it is impractical to simply keep adding lines with new sensor measurements. Instead, sensor measurements must be carefully integrated with previous measurements to produce an estimate of the wall's true position. Our method of combining data heavily weighted the LiDAR's incoming data, leading to significant information loss on every update. Improving this combination algorithm is an important step to improving continuous space mapping.

Another related issue was retaining old sensor measurements. Allowing the LiDAR to continuously capture and update the map while it was driving proved impractical: the data while it was moving was far more noisy than the data while it was stationary. This could also be fixed by improving the combination algorithm to use confidences in combining sensor data. Since this required a major restructuring, we simply turned off map updating while the robot was driving. This fix, while convenient, highlights the fundamental issue with the assumptions about sensor noise used for this project.

Beyond these issues, nearly every step of the continuous mapping required significant computation, largely by brute force. The low number of obstacles to check due to advantages of continuous mapping made these solutions viable, but adding multiple layers of higher-level planning would require optimization of the mapping algorithms. Using a more powerful computer, or ROS network capabilities, could also improve this performance.

VI. Conclusion

We developed a memory-efficient (though somewhat slow) mapping algorithm that operates in continuous space. Rather than fitting the world to a grid, our algorithm uses line segments to identify the exact position of obstacles. This enables highly precise maps of obstacle position with limited memory, and creates a framework ripe for higher-level planning.

Despite its advantages, our system is far from perfect. Working in continuous space complicates combining sensor measurements, retaining old sensor data, and identifying moving obstacles. The resulting maps were somewhat noisy and failed to accurately retain information about hidden walls. These problems, however, are not innate to a continuous mapping structure. Better handling of sensor noise, such as incorporating LiDAR data from multiple measurements and allowing the possibility of LiDAR missing an obstacle, could significantly improve the robot's ability to accurately retain information about obstacles as it moves. Additionally, a more robust occlusion checking system that incorporates the estimated walls as well as new sensor data could produce a much more accurate map.

Ultimately, this experiment in continuous mapping was mixed. Using continuous space to fit a map to noisy point cloud sensor measurements as the robot moved proved significantly more difficult than anticipated. The end result is an incomplete map, but one that may still be used for basic planning. With further improvements, a truly robust continuous mapping system is possible.